Du support générique d'opérateurs de composition dans les modèles de composants logiciels, application au calcul à haute performance.

Julien Bigot

Thèse

THESE INSA Rennes
sous le sceau de l'Université européenne de Bretagne
pour obtenir le titre de
DOCTEUR DE L'INSA DE RENNES
Spécialité : Informatique

présentée par
Julien Bigot
ECOLE DOCTORALE : *Matisse*
LABORATOIRE : IRISA - LIP

Du support générique d'opérateurs de composition dans les modèles de composants logiciels, application au calcul à haute performance.

Thèse soutenue le 06.12.2010
devant le jury composé de :

Antoine Beugnard
Professeur ENST de Bretagne / examinateur
Marco Danelutto
Associate Professor Université de Pise / rapporteur
Jean-François Méhaut
Professeur UJF en détachement INRIA / rapporteur
Olivier Morvant
Ing. de recherche chef de départ. délégué EDF / examinateur
Jean-Louis Pazat
Professeur INSA de Rennes / examinateur
Christian Pérez
Docteur, chargé de recherche INRIA, HDR / Directeur de thèse

Remerciements

Il y a trois ans, quand j'ai fait le choix de continuer en thèse, la rédaction semblait si loin ... Il y a un peu moins d'un an, quand j'ai commencé à rédiger, le moment où je taperai le point final me semblait encore bien abstrait. Aujourd'hui c'est fait, mais je n'y serais jamais arrivé tout seul. Il me reste donc à remercier tous ceux qui m'ont aidé. Ceux sans qui je ne pourrais pas profiter de la fierté de tenir ce manuscrit dans mes mains.

En premier lieu, je tiens à remercier les membres de mon jury qui m'ont fait l'honneur d'évaluer les travaux ici présentés. Je remercie Jean Louis Pazat, qui a présidé le jury. Je remercie Marco Danelutto et Jean-François Méhaut qui ont accepté de rapporter cette thèse dans des délais plus que serrés. Je remercie aussi Antoine Beugnard et Olivier Morvant qui ont pris part au jury. Je les remercie tous pour leur évaluation qui donne toute sa valeur à cette thèse ainsi que pour leurs remarques et questions pertinentes. Et bien sûr, je remercie Christian Pérez qui m'a encadré au cours de ces trois ans. Il a su me guider tout en me laissant une grande marge de manœuvre pour explorer mes idées les plus saugrenues.

Je tiens aussi à remercier les membres des deux équipes qui m'ont accueilli. Tout d'abord l'équipe PARIS à Rennes puis l'équipe GRAAL à Lyon. Je remercie d'abord les assistantes que j'ai côtoyé au sein de ces deux projets, Maryse, Caroline, Marie et Evelyne. Elles m'ont permis de me démêler des démarches administratives à côté desquelles l'informatique me semble parfois si simple. Je remercie aussi tous ceux que j'ai rencontré au cours de ces années et qui plus que des collègues sont devenus des amis. Ils sont trop nombreux pour que je les cite tous mais ils se reconnaîtront. Je leur donne rendez-vous au Tiffany's ou au Ninkasi. Je remercie finalement tous ceux avec qui j'ai eu de nombreuse discussions passionnantes et passionnées aux pauses café ou ailleurs.

Finalement, je remercie tous ceux qui ont été là pour moi au cours de ces trois ans et plus. Je pense bien sûr à ma famille, mais aussi à ceux qui font partie de la famille même si leur carte d'identité ne le précise pas. Même si je sais que beaucoup de ce que je raconte dans ces pages ne vous parle pas vraiment, vous m'avez toujours supporté dans mes choix. Je vous en remercie profondément. Je pense aussi à mes amis, qui m'ont aidé à penser à autre chose quand la rédaction me pesait trop. je tiens notamment à remercier Adhem, Amil, Éléonore, Éva, Jonathan, Raphaël, Simon et Vincent.

Enfin je tiens à remercier ceux que je n'ai pas manqué d'oublier dans cette pages mais que je n'oublie pas au fond de moi.

Table des matières

Table des figures

Chapitre 1

Introduction

Depuis ses débuts, l'outil informatique a été mis à contribution pour simuler l'exécution d'expériences qui seraient trop complexes, trop longues, trop coûteuses, trop dangereuses ou tout simplement impossibles à effectuer réellement. Si ces simulations numériques étaient initialement réservées à quelques projets de très grande ampleur comme le projet Manhattan, on les retrouve aujourd'hui dans tous les domaines. Elles sont par exemple utilisées par l'industrie pour la conception de carrosseries de voitures profilées, pour s'assurer que des bâtiments résisteront à des secousses sismiques, pour prévoir les effets de nouveaux médicaments ou pour vérifier la résistance de réacteurs nucléaires. Elles sont aussi utilisées par les scientifiques pour développer et valider leurs modèles théoriques dans de nombreux domaines comme par exemple la physique nucléaire, la génomique, l'astrophysique ou la météorologie.

Les simulations scientifiques constituent un véritable défi puisque les scientifiques tendent à intégrer de plus en plus de paramètres à leur simulations et à utiliser des granularités de plus en plus fines. Afin d'affiner encore les simulations, la tendance est à regrouper des modélisations des différents aspects de l'objet étudié au sein de simulations multi-physiques. C'est par exemple le cas des simulations météorologiques qui intègrent des simulations de nombreux aspects à l'échelle terrestre : température des océans et des masses d'air, circulation des courants marins et vents, influence de l'effet de serre et de la biosphère, fonte des glaces, etc.

Il faut toutefois noter que ces applications sont rarement développés en partant de zéro ; elles sont généralement issues du couplage de codes pré-existants et de nouveaux éléments conçus pour l'occasion. De ce fait, il n'est pas rare de voir des codes conçus et développés plusieurs décennies plus tôt (une éternité à l'échelle de l'informatique) continuer à être utilisés. La gestion des interactions entre de tels codes qui n'avaient pas initialement été prévus pour être utilisés conjointement peut s'avérer complexe. C'est d'autant plus problématique que les développeurs de ces codes n'ont pas à être experts en génie logiciel mais plutôt des spécialistes des divers domaines modélisés.

L'utilisation de modèles de plus en plus complexes a justifié et a été rendue possible par le développement de matériel permettant d'obtenir des puissances de calcul de plus en plus importantes, c'est à dire de traiter une quantité de données plus élevée par unité de temps. Cette augmentation de la puissance a d'abord été portée par l'augmentation de la fréquence à laquelle les opérations de traitement sont enchaînées par les processeurs. Pour améliorer encore les puissances obtenues, les opérations de traitement appliquées par les processeurs ont été modifiées pour s'appliquer à plusieurs données en même temps. Finalement, des machines qui regroupent plusieurs processeurs ont été développées pour agréger leur puissance de calcul.

Dans la mesure où les processeurs ne voient plus aujourd'hui leur fréquence augmenter régulièrement comme cela a longtemps été le cas, c'est sur la possibilité d'exécuter plusieurs flux d'instruction simultanément que se concentrent aujourd'hui les augmentation de performances. Même les processeurs des machines de bureau d'entrée de gammes possèdent aujourd'hui de multiples cœurs d'exécution et les machines plus haut de gamme regroupent

plusieurs processeurs. Dans ce contexte, deux approches distinctes peuvent être adoptées pour proposer des solutions permettant d'offrir les performances de calcul les plus élevées possibles :

- soit ce sont des super-calculateurs spécifiquement conçus pour le calcul à haute performance qui regroupent aujourd'hui jusqu'à plusieurs centaines de milliers de cœurs de calcul ;
- soit ce sont des machines plus classiques regroupant quelques processeurs pour un total de l'ordre de la dizaine de cœurs de calcul inter-connectées par des réseaux à haute performance au sein de grappes de calcul qui regroupent aujourd'hui jusqu'à plusieurs milliers de machines.

Avec l'omniprésence des réseaux à longue distance et l'amélioration de leur performances due à l'arrivée d'Internet, il est devenu intéressant de regrouper ces diverses ressources de calculs. Le concept de grille de calcul a été proposé par analogie avec la grille électrique pour mettre en relation les acteurs possédant des infrastructures de calcul de leurs utilisateurs. L'idée consiste à permettre à ces utilisateurs de soumettre leurs calculs sans avoir à se préoccuper du lieu ou de la manière dont ils sont exécutés. L'utilisation d'une telle architecture dans laquelle les acteurs impliqués, les architecture des ressources de calcul et leurs interconnexions sont hétérogènes s'avère complexe. On remarque que leur utilisation se restreint souvent aux cas les plus simples :

- soit l'application est massivement parallèle (*embarrassingly parallel*) et ne nécessite pas de communications entre les sous-tâches qui la compose qui peuvent donc aisément être réparties sur les différents sites de calcul ;
- soit l'application nécessite des communications entre ses différentes sous-tâches et l'utilisation de la grille consiste simplement à choisir l'un de ses sites de calcul pour y exécuter l'ensemble de l'application.

Plus récemment, les concepts de *cloud computing* (informatique dans le nuage) et plus particulièrement d'infrastructure comme un service (*Infrastructure as a Service* – IaaS) ont aussi été proposés. Il s'agit comme pour les grilles d'offrir la possibilité d'utiliser des ressources de calcul sans avoir à en faire l'acquisition grâce à des communications au travers d'Internet. Il faut cependant noter qu'il s'agit d'une approche plus pragmatique dans laquelle le client n'a affaire qu'à un unique fournisseur de ressources qu'il obtient généralement au sein d'un unique site de calcul relativement homogène. La notion de *sky computing* (informatique dans le ciel) qui consiste à agréger des ressources obtenues de plusieurs fournisseurs commence toutefois à émerger. Il semble donc que les différences entre *cloud* et grille vont continuer à aller en s'amoindrissant et que les problèmes étudiés dans le cadre de la grille vont devoir être résolus dans ce nouveau cadre.

L'exécution d'applications complexes comme peuvent l'être les simulations scientifiques et en particulier les simulations multi-physiques issues de couplage de codes sur des ressources de calcul complexes comme peuvent l'être les super-calculateurs ou les grilles de calcul pose un véritable défi. En effet, pour tirer pleinement parti de ces architectures, il est nécessaire de s'appuyer sur une connaissance de la plate-forme d'exécution et en particulier des coûts des calculs sur les divers éléments qui la composent ainsi que des coûts de communication entre ces éléments. Ainsi, il est habituel d'adapter les applications pendant la phase de construction d'un super-calculateur pour pouvoir s'adapter à ses spécificités.

Dans le cadre de la grille, l'hétérogénéité des ressources rend cette option impossible à envisager puisqu'il faudrait alors adapter les applications à l'ensemble des architectures matérielles qui apparaissent au cours de leur cycle de vie. À l'échelle de la grille, le problème est encore plus patent puisque la volatilité des ressources signifie que son architecture globale évolue à une échelle de temps qu'il convient de mesurer en minutes plutôt qu'en années.

Pour répondre à cette complexité, l'utilisation d'un modèle de programmation adapté est nécessaire. Une approche intéressante est offerte par les modèles à base de composants logiciels qui proposent de construire les applications par l'assemblage de composants dont les points d'interaction sont clairement identifiés. Cette approche favorise la réutilisation

et le développement indépendant des diverses parties des applications. De plus, pour permettre aux composants de se focaliser sur le code métier, il est fréquent que les aspects périphériques comme la sécurité ou la gestion de la distribution soient gérés par le modèle de manière transparente pour les composants. Un aspect qui n'est généralement pas géré concerne toutefois le domaine des hautes performances.

Afin d'adapter ce modèle de programmation au calcul à haute performance sur des ressources matérielles inconnues, des extensions ont été proposées. Il s'agit généralement d'intégrer au sein des modèles des opérations de composition d'un niveau d'abstraction plus élevé que les simples connexions point à point entre composants. Ces opérations de composition sont généralement inspirées par les paradigmes de programmation existant pour la gestion du parallélisme et de la distribution. La mise en œuvre de ces compositions peut alors être adaptée aux ressources matérielles effectivement utilisées.

Objectifs

Malgré leur intérêt, les modèles de programmation à base de composant pour le développement d'applications de calcul scientifique se limitent à un ensemble restreint de formes de composition. Le résultat est une tendance à la conception d'un nouveau modèle de composants pour chaque nouvelle application, ce qui met en échec l'objectif de réutilisation de code porté par ces modèles.

La thèse soutenue dans ce manuscrit est qu'*il est possible de concevoir un modèle de composants dans lequel de nouvelle formes de compositions peuvent être décrites*. Il s'agit de proposer un modèle qui permette non seulement d'utiliser les divers opérateurs de composition disponibles pour assembler les composants mais aussi de décrire de nouveaux opérateurs de composition. Il doit aussi être possible de décrire les mises en œuvre de ces opérations adaptées à diverses architectures de ressources matérielles.

Les objectifs qui sont poursuivis consistent donc à :

- permettre l'expression de l'ensemble des opérateurs de composition qui peuvent être nécessaire, ceux qui correspondent aux paradigmes de programmations existant mais aussi d'autres spécifiques à des classes restreintes d'application ;
- permettre l'utilisation conjointe au sein d'une même application des diverses opérations de composition nécessaires ;
- permettre une mise en œuvre efficace des opérations de composition pour un grand nombre d'architectures matérielles, y compris certaines qui n'existeraient pas encore au moment de la conception de l'application et assurer que la bonne version sera choisie.

Dans la mesure où il existe déjà un grand nombre de modèle de composants et que leur multiplication est problématique pour l'objectif de réutilisation, nous avons fait le choix de ne pas redéfinir un nouveau modèle de composants depuis zéro mais de s'appuyer sur l'existant. Afin de ne pas se restreindre à un modèle existant, nous avons aussi fait le choix de nous abstraire du modèle étendu pour proposer des solutions qui puissent être adaptées au plus grand nombre de modèles possible.

Contributions

Afin de permettre la description de nouveaux opérateurs de composition dans les modèles de composants, il est nécessaire d'y intégrer de nouveaux concepts. Après avoir identifié un ensemble d'opérateurs de composition qu'il est intéressant de supporter, nous avons proposé un ensemble de concepts qui permettent leur description :

hiérarchie la possibilité de partager des assemblages de composants partiels en les utilisant pour mettre en œuvre des composant qui sont alors dit composites ;

connecteurs la description des interactions entre les composants par des entités de première classe qui à la manière des composants peuvent être mis en œuvre dans le modèle ;

choix de mise en œuvre la possibilité pour les éléments de l'assemblage de posséder plusieurs mises en œuvre parmi lesquelles la plus adaptée est choisie en fonction des ressources matérielles utilisées ;

généricité la possibilité de décrire des éléments du modèle qui en acceptent d'autre comme paramètres.

L'ensemble de ces concepts forment la base du *High Level Component Model* (HLCM) que nous proposons. Cependant comme nous l'avons noté, nous avons essayé de nous abstraire du modèle de composants auquel ces concepts sont ajouté. Chaque combinaison des concepts d'un modèle de base et de ceux apportés par HLCM forment ainsi une *spécialisation* de HLCM comme par exemple HLCM/CCM qui étend le modèle de composants de CORBA (CORBA *Component Model* – CCM) avec les concepts de HLCM.

Si l'intégration de certain de ces concepts au sein d'un modèle de composants relève de l'ingénierie, certains aspects sont plus problématiques. La définition du concept de généricité appliqué aux composants et sa mise en œuvre doivent être précisément spécifiées. L'interaction entre les concepts de hiérarchie et de connecteurs pose problème quand l'approche classique pour l'introduction du concept de connecteur est adoptée.

Les contributions présentés dans ce manuscrit sont les suivantes.

Introduction de la généricité En combinaison avec la hiérarchie, le concept de généricité intégré aux modèles de composants permet de décrire des structures de composition dans lesquelles le traitement à appliquer ou le type des données traitées est un paramètres. Il s'agit en fait des concepts qui permettent la description de squelettes algorithmiques.

Connecteurs et hiérarchie Si le concept de connecteur permet de découpler la mise en œuvre du code fonctionnel des interactions entre ces codes, en présence de la hiérarchie l'approche habituellement utilisée pour l'intégrer aux modèles de composants pose problème. Elle présente une lacune qui peut empêcher de concilier performances optimales et interchangeabilité des mises en œuvres de composant. Nous avons donc proposé une nouvelle approche pour la description des interactions entre les composants basée sur le concept de *connexion ouverte* pour résoudre ce problème.

Plate-forme de mise en œuvre Afin d'utiliser le modèle ainsi défini, nous avons développé une plate-forme pour sa mise en œuvre nommée HLCM *implementation* (HLCM_i) qui est elle même développée à l'aide de composants. Elle offre une base pour la mise en œuvre des diverses spécialisations de HLCM. Pour cela, elle s'appuie sur une approche inspirée de l'ingénierie basée sur les modèles (*Model-Driven Engineering* – MDE). Nous avons utilisé cette plate-forme pour mettre en œuvre HLCM/CCM.

Évaluation Le modèle HLCM et plus particulièrement sa spécialisation HLCM/CCM ainsi que sa mise en œuvre basée sur HLCM_i ont été évalués en les utilisant pour mettre en œuvre des applications synthétiques. Ces applications constituées de couplages de codes pouvant être parallèles à l'aide d'interactions par partage de donnée et par appel de méthodes parallèles ont été développée en s'appuyant sur HLCM/CCM et exécutées sur la plate-forme expérimentale *Grid'5000*. Le modèle HLCM et la plate-forme de mise en œuvre HLCM_i ont aussi été utilisés pour permettre l'adaptation aux ressources d'exécution d'applications basées sur des opérations de communication collective hiérarchiques.

Organisation de ce document

Ce document s'articule en huit chapitres. Après cette introduction, le chapitre 2 présente le contexte de notre étude. Les paradigmes de programmation destinés à tirer parti des architectures parallèles et distribuées y sont présentés ainsi que les modèles de programmation par composition destinés à faciliter le développement d'application par la réutilisation de composants existants. Finalement, les travaux destinés à adapter les modèles de composants logiciels au calcul à haute performance en permettant l'utilisation de paradigmes de programmation dédiés sont analysés et leur adéquation avec les besoins pour le développement d'applications de simulation discutée.

Le chapitre 3 présente notre approche pour la conception d'un modèle de programmation à base de composants permettant la spécification de nouvelles opérations de composition. Les objectifs à atteindre y sont définis et les travaux existants analysés et discutés. Notre approche est détaillée et les concepts de l'ingénierie des modèles sur laquelle elle s'appuie sont présentés.

Le chapitre 4 présente notre première contribution qui consiste à intégrer la généricité aux modèles de composants pour y permettre la description de squelettes algorithmiques. Une analyse préliminaire explicite le lien entre généricité et squelettes algorithmiques au travers d'un exemple motivant et évalue les travaux existants dans ce domaine. Notre proposition est ensuite discutée et présentée en décrivant les nouveaux concepts qu'elle nécessite et en spécifiant leur comportement. Une validation préliminaire est proposée qui s'appuie sur une application à l'architecture de composant de service (*Service Component Architecture* – SCA) pour former GenericSCA.

Le chapitre 5 présente notre seconde contribution qui consiste en une nouvelle approche pour la description d'instances de connecteurs afin d'améliorer les performances obtenues et l'adaptabilité aux ressources d'exécution. Une analyse préliminaire présente les approches existantes et explicite le problème qu'elles posent au travers d'un exemple. Notre proposition qui constitue le modèle HLCM est ensuite discutée et présentée. Les nouveaux concepts de connexion ouverte, d'adaptateur de connexion et de port *bundle* qu'elle nécessite sont présentés et leur comportement spécifié. Une validation préliminaire est proposée qui s'appuie sur une application au modèle de composants de Corba (Corba *Component Model* – CCM) pour former HLCM/CCM.

Le chapitre 6 présente notre troisième contribution qui consiste en une plate-forme nommée HLCM_i pour la mise en œuvre de spécialisations de HLCM et son utilisation pour la mise en œuvre de HLCM/CCM. Les technologies de composants et de modélisation sur lesquelles s'appuie cette plate-forme sont tout d'abord présentées. Les éléments de la plate-forme communs aux différentes spécialisations sont ensuite décrits et leur utilisation pour mettre en œuvre HLCM/CCM étudiée.

Le chapitre 7 propose une évaluation de nos travaux. L'utilisation de HLCM et en particulier de HLCM/CCM pour la description d'interactions entre des codes pouvant être parallèles par mémoire partagée et par appels de méthodes est analysée. L'utilisation du prototype dans le cadre d'un modèle de composition spécifique basé sur des opérations de communication collective hiérarchique pour lequel des heuristiques de choix avancées existent est aussi étudiée.

Finalement, le chapitre 8 revient sur l'ensemble de cette étude pour en tirer les conclusions et propose un aperçu de travaux qui s'inscrivent dans sa perspective.

Publications

Publications dans des conférences internationales

[BBPP09] Julien Bigot, Hinde Lilia Bouziane, Christian Pérez, and Thierry Priol. On abstractions of software component models for scientific applications. In Eduardo César, Michael Alexander, Achim Streit, Jesper Larsson Träff, Christophe Cérin,

Andreas Knüpfer, Dieter Kranzlmüller, and Shantenu Jha, editors, *Euro-Par 2008 Workshops - Parallel Processing*. Springer-Verlag, Las Palmas de Gran Canaria Spain, 04 2009. doi :10.1007/978-3-642-00955-6.

[BP09] Julien Bigot and Christian Pérez. Increasing reuse in component models through genericity. In Stephen H. Edwards and Gregory Kulczycki, editors, *Proceedings of the 11th International Conference on Software Reuse, ICSR '09 Formal Foundations of Reuse and Domain Engineering*. Springer-Verlag, Falls Church, VA United States, 09 2009. doi :10.1007/978-3-642-04211-9.

[BP07] Julien Bigot and Christian Pérez. Enabling collective communications between components. In *CompFrame '07 : Proceedings of the 2007 symposium on Component and framework technology in high-performance and scientific computing*. ACM Press, Montreal, Quebec Canada, 2007. doi :10.1145/1297385.1297406.

Chapitre de livre en cours

[BP11] Julien Bigot and Christian Pérez. *High Performance Scientific Computing with special emphasis on Current Capabilities and Future Perspectives*, chapter On High Performance Composition Opertators in Component Models. Advances in Parallel Computing. IOS Press, 2011. Invited paper from the HPC2010 workshop at Cetraro, Italy, 21–25 June 2010.

Articles en cours d'évaluation pour publication dans des conférences internationales

[BP11] Julien Bigot and Christian Pérez. Enabling connectors in hierarchical component models. In *ICSE'11 Proceedings of the International Conference on Software Engineering 2011*. 2011.

Rapports de recherche

[RR_BP10] Julien Bigot and Christian Pérez. Enabling connectors in hierarchical component models. Technical report, INRIA, 2010.

[RR_BP09a] Julien Bigot and Christian Pérez. A generic model driven methodology for extending component models. Technical report, INRIA, 2009.

[RR_BP09b] Julien Bigot and Christian Pérez. Increasing reuse in component models through genericity. Technical report, INRIA, 2009.

CHAPITRE 2

Contexte d'étude

Les applications de calcul scientifique nécessitent de par leurs spécificités l'utilisation de modèles de programmation adaptés. Ces modèles doivent répondre à deux besoins : faciliter le développement des applications par l'assemblage d'éléments de multiples origines et permettre leur exécution efficace sur des ressources matérielles parallèles et réparties diverses. Ce chapitre présente des modèles comportant des aspects intéressants pour le développement de ces applications.

La section 2.1 présente des paradigmes de programmation adaptés au calcul à haute performance et la section 2.2 des modèles de programmation par composition. La section 2.3 s'attarde ensuite sur les travaux qui proposent de rapprocher ces deux mondes en permettant aux paradigmes pour la programmation distribuée et parallèle d'être utilisés au sein de modèles de composants logiciels. Finalement, la section 2.4 conclue le chapitre.

2.1 Des paradigmes de programmation pour le calcul à haute performance

Plusieurs paradigmes ont été proposés pour faciliter la gestion du grand nombre de flux d'exécution parallèles nécessaire à l'obtention de hautes performances sur les ressources matérielles utilisées dans le cadre du calcul scientifique. Ces paradigmes peuvent être classifiés en deux catégories suivant qu'ils proposent aux flux d'exécution qui composent l'application la vision d'un espace mémoire unique auquel ils accèdent tous ou d'espaces mémoire distincts pour chacun.

2.1.1 Mémoire partagée

L'accès à un espace mémoire commun par les différents flux d'exécution leur permet d'interagir par l'écriture et la lecture de zones de mémoire convenues. La principale difficulté qui apparaît lors de l'utilisation de cette approche concerne la cohérence des données présentes en mémoire. En effet, l'ensemble des opérations qui constituent une transition entre deux états cohérents ne peut être connu que par le logiciel.

Programmation concurrente avec verrous Une première approche consiste à laisser entièrement aux applications la gestion de ce problème en leur fournissant des mécanismes permettant d'assurer l'exclusion de l'accès aux données. Le mécanisme le plus couramment utilisé est constitué par les verrous d'exclusion mutuelle (*mutex*) qui permettent d'assurer qu'un unique flux d'exécution possède le verrou à un instant donné en bloquant l'exécution des autres flux qui tentent de l'obtenir pendant ce temps.

À l'échelle d'une machine possédant une mémoire matériellement partagée, ces mécanismes sont offerts par le système d'exploitation et accessibles depuis des langages comme le C ou le C++ via des bibliothèques partagées. La norme Posix [36] (*Portable Operating System Interface [for uniX]*) spécifie notamment l'interface et le comportement d'une bibliothèque de ce type pour les systèmes Unix. Dans d'autres langages, les mécanismes de synchronisation sont intégrés au langage. Par exemple en Java, le concept de moniteur [35] utilisable à l'aide du mot clef *synchronized* revient à associer un verrou à chaque objet.

Au sein de grappes de calcul ou *a fortiori* de grilles pour lesquelles il n'existe pas de mémoire partagée au niveau physique, des mécanismes logiciels doivent être mis en place pour offrir une abstraction similaire. On parle de mécanismes de mémoire partagée distribués [54] (*Distributed Shared Memory* – DSM). Au sein de grappes de calcul qui s'appuient sur des réseaux à haute performance certaines mises en œuvre s'intègrent au système d'exploitation. Elles permettent de conserver la même sémantique de cohérence de données qu'une mémoire physiquement partagée. C'est par exemple ce que propose *Kernel Distributed Data Management* [39] (kDDM). Le coût de cette approche est toutefois relativement élevé et devient totalement prohibitif à l'échelle de la grille.

Pour résoudre ce problème de performances, une approche consiste à s'appuyer sur un transfert d'information plus important de l'application vers le mécanisme de mémoire partagée distribué. C'est par exemple l'approche adoptée par JuxMem [5] que nous avons utilisé dans nos travaux et qui s'appuie sur l'association d'un verrou à chaque zone de mémoire partagée qui doit être acquis avant tout accès aux données.

L'accès à un espace mémoire unique partagé est intéressant pour les applications dans lesquelles il n'est pas possible de répartir les données traitées entre les flux d'exécution. C'est par exemple le cas des applications qui mettent en œuvre des modèles où la notion de localité a peu d'importance. Par exemple pour simuler l'éclairement d'une scène par des techniques de lancer de rayon, une approche consiste à faire gérer par chaque flux d'exécution un ensemble de photons. Les diverses réflexions et réfractions sur les objets impliquent que ces photons peuvent avoir des effets sur l'ensemble de la scène. Chaque flux d'exécution est donc susceptible d'accéder à l'ensemble des données sans qu'il soit possible d'identifier de motif régulier et il n'est donc pas aisé de proposer une répartition des données *a priori* entre eux.

La gestion manuelle des interactions entre divers flux d'exécution qui accèdent à un même espace mémoire peut cependant s'avérer complexe. Elle nécessite d'étudier tous les risques d'inter-blocages qui peuvent apparaître dès que les flux d'exécution sont susceptibles d'acquérir plusieurs verrous à un instant donné. Quand le nombre de flux d'exécution augmente, cette complexité impose rapidement de se tourner vers des modèles offrant des mécanismes plus adapté.

Espace mémoire global partitionné Afin d'éviter de devoir gérer manuellement l'exclusion mutuelle pour les accès au donné, un paradigme intéressant est offert par le concept

d'espace d'adressage mémoire global partitionné (*Partitioned Global Address Space* – PGAS). Ce paradigme propose de laisser les données accessibles depuis l'ensemble des flux d'exécution mais de répartir leur traitement entre les flux lors de phases parallèles.

Ce principe se traduit généralement au niveau des langages par la présence de tableaux pouvant être manipulés par ces boucles parallèles. Il s'agit d'appliquer un traitement sur le contenu des tableaux en répartissant les itérations sur les différents flux d'exécutions. Entre chaque section parallèle, une synchronisation assure que tous les flux d'exécution ont terminé leur traitement

De nombreuses mises en œuvre de ce paradigme ont été proposées. Certaines sont des extensions au langage Fortran comme par exemple *High Performance* Fortran [34] (HPF) ou *Co-array* Fortran [46] (CAF ou F--) dont une partie des extensions a été intégrée au standard Fortran95. Un autre langage qui met en œuvre ce paradigme mais qui est basé sur C est *Unified Parallel* C [27] (UPC). Finalement, OpenMP [51] propose un ensemble d'annotations qui peuvent être ajoutées à des programmes en C, C++ et Fortran.

D'autre part, les langages proposés pour répondre au projet de systèmes de calcul à haute productivité (*High Productivity Computing Systems* – HPCS) de l'agence *Defense Advanced Research Projects Agency* (Darpa) aux États-Unis que sont X10 [23], Chapel [22] et Fortress [28] s'appuient aussi à divers degrés sur ce paradigme.

Ce type d'approche convient bien à des codes où le parallélisme peut être obtenu en répartissant l'exécution des boucles de traitement de données sur les différents cœurs d'exécution des machines. Contrairement à l'utilisation de verrous d'exclusion mutuelle précédemment évoquée, cette approche repose sur une répartition des donnés entre les flux d'exécution mais cette répartition peut changer au cours de l'exécution de l'application C'est par exemple le cas quand les boucles que contient l'application itèrent sur des dimensions différentes de tableaux multi-dimensionnels.

Les sections parallèles entre deux synchronisations restent cependant généralement relativement courtes et fréquentes. Ce mécanisme est donc bien adapté à des architecture où les coûts de communication entre cœurs d'exécution sont faibles.

2.1.2 Mémoire distribuée

Lorsque chaque flux d'exécution évolue dans son propre espace mémoire, la coopération entre les différents flux pour mettre en œuvre l'application doit s'appuyer sur des échanges explicites d'information. Différents paradigmes offrent des abstractions de plus ou moins haut niveau pour envoyer et recevoir ces messages.

Passage de messages Le niveau d'abstraction le plus bas consiste à exposer directement à l'application la possibilité d'envoyer et de recevoir des messages comme le supporte le réseau sous-jacent. C'est ce que proposent par exemple les *sockets* Unix en permettant l'envoi de flux d'octets à destination d'une adresse réseau spécifique.

À un niveau d'abstraction à peine plus élevé, des bibliothèques comme *Parallel Virtual Machine* [30] (PVM) et *Message Passing Interface* [58, 33] (MPI) proposent de masquer les aspects spécifiques au réseau utilisé. Le destinataire de chaque message peut par exemple y être identifié par son rang dans l'ensemble des flux qui forment l'application plutôt que par une adresse réseau. Ces envois peuvent être bloquant, mais ils peuvent aussi être non-bloquants pour leur permettre d'avoir lieu pendant que les calculs continuent.

Ces échanges de messages point à point entre couples de flux d'exécution deviennent vite compliqués à gérer dès que le nombre de flux d'exécution augmente. Leur efficacité dépend de l'architecture des réseaux et leur optimisation doit prendre en compte les capacités des liens qui connectent les différentes ressources, tant en terme de débit que de latence. Afin de faciliter le développement d'applications pouvant s'exécuter efficacement sur un plus grand nombre de ressources, des interactions offrant un plus haut niveau d'abstraction sont nécessaires.

Opérations de communication collective Les opérations de communication collective proposent des schémas d'interaction classiques au sein de groupes de flux d'exécution. Un exemple simple de ce type d'opération est la barrière (*barrier*) qui permet une synchronisation entre les flux d'exécution du groupe en assurant qu'ils ont tous atteint un même point dans leur exécution avant que celle-ci reprenne. Les autres opérations permettent des échanges de données au sein du groupe, on peut par exemple citer :

- la diffusion (*broadcast*) qui consiste à répliquer une donnée d'un flux donné (la racine) vers l'ensemble des membres du groupe ;
- la répartition (*scatter*) qui consiste à répartir une donnée de la racine vers tous les membre du groupe ;
- la réduction (*reduce*) qui peut être considérée comme l'opération inverse de la diffusion ; elle consiste, à partir de données réparties sur tous membres du groupe, à construire une unique donnée sur la racine ; celle-ci est obtenue par l'application d'une opération de réduction pour regrouper plusieurs valeurs en une seule ;
- le rassemblement (*gather*) qui peut être considéré comme l'opération inverse de la répartition ; il consiste à rassembler des données réparties sur l'ensemble des membres du groupe vers la racine.

Les deux principales mises en œuvre de ce paradigme sont offertes par PVM et par MPI. Elles s'appuient sur une bibliothèque qui offre les diverses opérations de communication point à point et collectives qui peuvent être utilisées. Pour créer les divers flux d'exécution, elles s'appuient sur un programme spécifique qui doit être utilisé pour lancer l'application et auquel les informations sur le matériel sont fournies. Une autre approche adoptée par exemple pour Charm++ [37] propose d'intégrer ce paradigme directement au sein du langage de programmation.

L'utilisation d'opérations de communication collective permet aux applications d'exprimer les interactions entre flux d'exécution à un plus haut niveau d'abstraction que de simples opérations point à point. Leur mise en œuvre au sein de bibliothèques permet de proposer des versions adaptées au matériel qui prennent en compte les possibilités des réseau pour éviter la contention et tirer parti au maximum des liens les plus rapides. Ces mises en œuvre peuvent être mises à jour pour s'adapter aux évolutions du matériel même après que l'application ait été développée. De plus, la spécification d'opérations de communication au sein d'un groupe dont la taille n'est pas spécifiée permet de faire varier le nombre de flux d'exécution utilisés en fonction du nombre de cœurs d'exécution disponibles.

Ces opérations sont particulièrement utilisées pour les applications de type *Single Program, Multiple Data* (SPMD). Il s'agit d'applications formées d'un ensemble de flux d'exécution qui exécutent tous le même algorithme sur des données différentes. Ces applications s'appuient sur des échanges entre ces flux selon des motifs réguliers pour lesquels les opérations de communication collective sont particulièrement adaptées.

Appels de procédure à distance Une variation de cette approche est offerte par les modèles qui proposent un concept d'appel de procédures à distance [13] (*Remote Procedure Call* – RPC) ou d'invocation de méthodes à distance (*Remote Method Invocation* – RMI). Il s'agit de permettre à un flux d'exécution de transmettre des données à un second sur lequel un traitement est déclenché pour finalement renvoyer le résultat à l'appelant.

Ces appels peuvent offrir une sémantique très proche des appels de procédure locaux en bloquant l'exécution de l'appelant pendant le calcul par le flux d'exécution appelé. Dans le cas où le cœur de calcul qui héberge le flux d'exécution effectuant l'appel n'en possède pas d'autre en attente, cette approche n'apporte cependant aucun avantage en terme de parallélisme. Pour contourner cette difficulté, une autre approche consiste à faire évoluer la sémantique des appels de méthodes vers des appels non bloquants. Dans la mesure où les deux flux d'exécution (appelant et appelé) s'exécutent alors simultanément, cette option est cependant complexe à utiliser dans le cas de procédures renvoyant un résultat à la fin de leur exécution.

Pour résoudre ce problème, le concept de *future* a été intégré à certains modèles. Un *future* est une variable qui représente une donnée dont le calcul a été lancé mais dont la valeur n'est pas forcément encore disponible. Un *future* peut être utilisé comme n'importe quelle autre variable, mais l'accès à sa valeur bloque l'exécution du flux d'exécution jusqu'à ce que le résultat du calcul soit reçu.

Les appels de procédures et de méthodes à distance sont utilisables depuis la grande majorité des langages par l'utilisation de bibliothèques qui s'intègrent à des niveaux plus ou moins élevés avec les environnements de développement. On peut par exemple citer les RPC de l'*Open Network Computing* [59] (ONC RPC) pour le langage C qui sont notamment utilisées au sein de l'interface du *Network File System* (NFS). Les XML-RPC spécifient un protocole qui est mis en œuvre par diverses bibliothèques pour des langages tels que C, C++, Java, Python, etc. Les Java RMI font partie de la spécification du langage Java et permettent des appels de méthodes sans nécessiter de sérialisation explicite des données transmises. Corba [48] propose une fonctionnalité similaire indépendamment du langage en s'appuyant sur une description des interfaces dans un langage spécifique l'*Interface Description Language* (IDL) pour générer le code correspondant dans le langage cible.

Le concept de *future* est lui disponible dans la bibliothèque standard du langage Java et prévu pour la prochaine version de C++ [1]. Il est aussi directement intégré dans des langages pour permettre de l'utiliser de manière transparente, par exemple au sein de ProActive [10].

L'appel de procédure à distance offre de par sa ressemblance avec les appels locaux un paradigme simple à appréhender pour coupler des codes s'exécutant dans des espaces mémoires différents. Il faut cependant noter que contrairement à la majorité des approches, ce paradigme résulte en une répartition de la charge de calcul sur les différents cœurs d'exécution en fonction des traitements appliqués plutôt qu'en fonction des données traitées. Il peut s'agir d'un point positif si les ressources sont hétérogènes et que leur efficacité pour l'exécution des différents types de traitement varie. À l'inverse, il peut s'agir d'un véritable problème si les traitements sont appliqués de manière séquentielle aux données puisque dans ce cas, les différents cœurs d'exécution ne peuvent pas être utilisés simultanément.

L'introduction du concept de *future* permet d'obtenir un certain degré de parallélisme sans intervention spécifique de la part du développeur. Le coût de ces mécanismes qui nécessitent de sérialiser les données pour les transmettre au travers du réseau à chaque appel les restreignent cependant à des couplages à grain moyen.

Dans la pratique, ce paradigme est souvent utilisé au sein des applications de calcul scientifique pour le couplage de codes parallèles qui s'appuient sur une mémoire partagée ou sur une bibliothèque de passage de messages comme MPI. Dans ce cas, les appels de méthodes point à point entre processus sont complexe à mettre en œuvre efficacement.

Appel parallèle de procédure Pour résoudre cette difficulté, le concept d'appel parallèle de procédure ou de méthode propose une approche qui est à l'appel de procédure à distance ce que les opérations de communication collective sont au passage de message. Il s'agit de permettre à un groupe de flux d'exécution qui s'exécutent en parallèle d'effectuer un appel de procédure vers un autre groupe de flux d'exécution. Pour cela, au sein de chaque flux d'exécution du groupe appelant, une procédure est appelée avec en paramètre la partie des données gérée par ce flux. Chaque flux du groupe appelé reçoit alors un appel comportant en paramètre la partie des données qu'il doit gérer.

Dans le cas où les deux groupes de flux d'exécution s'appuient sur des distributions différentes pour les données ou si ils comportent un nombre différent de flux d'exécution, il est nécessaire que les données soient redistribuées. On parle alors d'appel de méthode $M \times N$. Cela implique qu'une méthode pour spécifier la distribution dans au sein de chaque groupe doit être offerte.

L'appel parallèle de méthode est par exemple disponible au sein de *PArallel COrba* [53] (Paco++) sous la forme d'une extension de Corba.

L'appel de méthode parallèle permet —à l'image des opérations de communication collective— de tirer parti des possibilités du matériel sans nécessiter que celui-ci soit connu des développeurs de l'application. Tout comme les appels de méthode classiques, il propose un paradigme simple à appréhender mais qui est adapté au couplage de codes à grain relativement gros qui sont déjà parallèles et doivent pour cela s'appuyer sur d'autres paradigmes.

Paradigme maître/travailleurs Une autre variante de l'appel de méthode est offerte par le paradigme maître travailleurs. Il s'agit d'offrir à un flux d'exécution particulier (le maître) la possibilité d'effectuer des appels de méthodes dont le traitement est effectué par l'un des flux d'exécution d'un ensemble de travailleurs. Le choix du travailleur à utiliser est du ressort de l'environnement de programmation qui met en œuvre le paradigme et généralement automatisé pour optimiser un objectif préalablement choisi comme par exemple la quantité de requêtes traitées par unité de temps.

Ce paradigme est mis en œuvre par diverses mises en œuvres parmi lesquelles on peut citer à l'échelle de la grille celles qui s'appuient sur le standard GridRPC [57] de l'*Open Grid Forum* (OGF). Il s'agit d'une interface de programmation (*Application Programming Interface* – API) qui est mise en œuvre par des outils comme la *Distributed Interactive Engineering Toolbox* [20] (Diet), NetSolve [8] ou Ninf [55]. Les appels y sont dissociés en deux étapes distinctes :

- une première étape consiste à effectuer une requête auprès de l'outil pour obtenir le travailleur à utiliser ; et
- la seconde étape consiste à faire la requête proprement dite auprès du travailleur.

Le paradigme maître/travailleurs est particulièrement intéressant dans le cadre d'applications qui exécutent un nombre élevé d'instances indépendantes d'un même code avec des valeur différentes des paramètres. C'est par exemple le cas des applications qui explorent l'espace des paramètres d'un modèle (*parameter sweep*) par exemple pour déterminer la combinaison de paramètres qui permettent au modèle de correspondre au plus près à la réalité.

2.1.3 Analyse

Les paradigmes utilisés pour le développement d'applications de calcul scientifiques sont nombreux. Une caractéristique qu'ils offrent fréquemment pour permettre d'obtenir des performances élevées sur diverses architectures matérielles consiste à élever le niveau d'abstraction pour l'expression des applications. En se rapprochant d'une description de l'objectif visé (le quoi) plutôt que de l'enchaînement d'opérations nécessaires à cet objectif (le comment), il devient possible d'utiliser des mises en œuvres différentes ou d'en faire varier des paramètres pour s'adapter aux ressources matérielles utilisées.

Du point de vue des mises en œuvres, deux approches ont été rencontrées :

- soit le paradigme est géré au sein des langages existants par une bibliothèque elle-même exprimée à l'aide de ce langage ;
- soit il est géré par un langage dédié qui peut constituer une extension d'un langage existant.

Les mises en œuvres qui s'appuient sur des bibliothèques sont limitées par les concepts qui sont exprimables dans le langage cible. À l'inverse, le développement de langages dédiés rend complexe l'utilisation simultanée de plusieurs paradigmes au sein d'une même application.

Les différentes approches proposent cependant de mettre le paradigme utilisé au centre de l'application. C'est par exemple le cas des mécanismes d'appels de méthode à distance qui nécessitent l'utilisation des types de données qu'ils fournissent. C'est aussi le cas des mécanismes de passage de message qui nécessitent que les divers flux d'exécution soient créé par leur biais.

Il en résulte qu'il est complexe d'utiliser plusieurs mécanismes et donc plusieurs paradigmes au sein d'une même application indépendamment de l'approche adoptée pour leur

mise en œuvre. Cet état de fait rend aussi complexe l'analyse de l'architecture des application et la réutilisation d'une partie de ces codes au sein de nouvelles applications.

2.2 Des modèles de programmation par composition

La notion de programmation par composition vise à rendre explicite l'architecture des applications et à simplifier leur développement à partir d'un ensemble d'éléments d'origines diverses. L'idée n'est pas neuve puisque le concept de composant logiciel a été proposé en 1968 par Douglas McIlroy [44]. Depuis ces origines, ce concept est toutefois resté relativement flou quant à ses propriétés exactes. Aujourd'hui, une définition qui fait à peu près consensus est proposée par Clemens Szyperski :

> «A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. [1]» [62]

Cette définition est suffisamment large pour inclure des concepts qui peuvent aller des bibliothèques partagées de procédures aux services *web*. Nous avons fait le choix de restreindre notre étude aux modèles qui séparent le développement des applications en deux étapes distinctes :

- une première étape où des éléments qui permettent d'appliquer divers traitement aux données qui constituent l'application sont développés ; et
- une seconde étape où ils sont assemblés pour former l'application.

Ce choix met de côté les modèles qui ne font pas de distinction claire entre ces deux étapes ou dans lesquels l'assemblage n'est pas explicite.

Trois modèles qui correspondent à cette définition sont présentés dans cette section. Les deux premiers proposent de composer les éléments qui forment l'application selon deux dimensions différentes. Les modèles de flux de données et de travail proposent un assemblage selon la dimension temporelle avec un enchaînement de tâches appliquant des traitements aux données. Les modèles de composants logiciels proposent un assemblage selon la dimension spatiale avec une exécution simultanée de composants qui s'échangent des informations. Un troisième modèle propose une approche différente qui s'appuie sur un ensemble de schémas de composition prédéfinis : les squelettes algorithmiques.

2.2.1 Les flux de données et de travail

flux de données Les modèles de flux de données (*dataflow*) proposent de découper les applications en un ensemble de *tâches* autonomes qui traitent les données qui leur sont fournies en entrée pour générer d'autres données en résultat. Les points d'interaction de ces *tâches* sont appelés ports et sont généralement nommés et typés par le type de la donnée attendue ou générée. Chaque *tâche* est donc définie par son interface qui correspond à l'ensemble de ses ports et par le code qui met en œuvre le traitement qu'elle applique.

Les applications sont construites par l'assemblage d'un ensemble d'instances de *tâches*. Les entrées de chacune y sont mises en correspondance soit avec des données initiales de l'application soit avec les sorties d'une autre *tâche*. Pour qu'un tel assemblage soit valide, l'introduction d'une quantité finie de données en entrée ne doit pas donner lieu à une durée d'exécution infinie, c'est à dire que le chemin des données ne doit pas former de boucles. Pour cela, l'assemblage doit être porté par un graphe dirigé sans cycles (*Directed Acyclic Graph* – DAG).

L'exécution d'un tel assemblage est généralement délégué à un moteur d'exécution. Ce moteur d'exécution peut s'appuyer sur une connaissance des ressources matérielles pour estimer le coût d'exécution de chaque tâche au sein de chaque cœur d'exécution et du

1. Un composant logiciel est une unité de composition dont toutes les interfaces et dépendances au contexte sont spécifiées contractuellement. Un composant logiciel peut être déployé indépendamment et être composé par des parties tierces.

transfert des données entre les cœurs. Il peut ainsi effectuer les choix de placement qui minimisent ces coûts et permettent d'optimiser l'exécution de l'application. Dans la mesure où les *tâches* ne sont généralement pas sensées posséder d'état interne, il est aussi possible à ces moteurs d'utiliser plusieurs cœurs d'exécution pour exécuter simultanément une même *tâche* logique sur des données différentes.

Ce modèle est cependant limité aux cas où les traitements appliqués aux données ne dépend pas de ces dernières. Il existe des situations où il est intéressant de pouvoir adapter les traitements aux données traitées. On peut par exemple vouloir n'appliquer un filtre sur des données que dans le cas où elles sont trop bruitées pour être utilisées sans cette étape. On peut aussi vouloir appliquer un traitement un nombre indéfini de fois jusqu'à ce qu'un critère de convergence soit rempli.

Flux de travail Ces besoins sont pris en compte par les modèles de flux de travail (*workflow*) qui proposent de décrire les applications par la combinaison d'un flux de données et d'un flux de contrôle. Le flux de données permet comme précédemment de spécifier quelles données doivent être traitées par chaque *tâche*. C'est cependant le flux de contrôle qui définit quelles sont les tâches qui s'exécutent.

Ce flux de contrôle peut s'appuyer sur des structures conditionnelles permettant d'exécuter une *tâche* ou une autre suivant la valeur des données. Il devient ainsi possible d'appliquer des traitements différents en fonction des données. Un cas particulier est constitué par la situation où les données en entrée d'une *tâche* donnée sont obtenue par l'exécution précédente de cette même *tâche* pour former une boucle. Contrairement au cas des flux de données purs, cette situation n'est pas problématique puisque la boucle peut être interrompu par les conditions portées par le flux de contrôle.

Parmi les modèles de flux de travail dédiés au calcul scientifique, on peut par exemple citer *Abstract Grid Workflow Language* (AGWL) utilisé au sein de l'environnement Askalon, Triana utilisé au sein de l'environnement *Grid Application Toolkit* (GAT) ou Kepler qui s'appuie sur l'environnement Ptolemy II.

Certains modèles comme AGWL sont hiérarchiques, c'est à dire que les *tâches* peuvent y être définies par un sous flux de travail. La spécification des modèles pour la définition des *tâches* primitives et de leurs interactions varie aussi suivant les modèles. Les applications peuvent généralement être des applications ou des services *web* qui communiquent soit à l'aide de protocoles dédiés soit par l'échange de fichiers.

Certains modèles comme Triana intègrent la notion de *tâche* locale pour les traitements qui nécessitent peu de puissance de calcul et qui bénéficient donc d'être exécutés au sein du même flux d'exécution que le moteur d'exécution. Pour la description des flux de travail, certains modèles s'appuient sur un langage dédié, souvent basé sur XML et d'autres proposent d'utiliser des outils graphiques.

2.2.2 Les modèles de composants logiciels

Les modèles de composants logiciels proposent de développer les applications en assemblant des composants qui exposent les services qu'ils fournissent et ceux sur lesquels ils s'appuient. Ces points d'interaction sont appelés ports et sont généralement nommés et typés par une interface objet. Chaque composant est donc formé d'une interface qui comprend l'ensemble de ses ports (et donc autant d'interfaces objets) et du code qui met en œuvre les méthodes des interfaces objets qu'il fournit en s'appuyant sur celles qu'il utilise.

Les applications sont construites par l'assemblage d'instances de composants. Les ports qui représentent un service requis (que nous appelons ports `uses`) y sont connectés à un port représentant un service fournit (que nous appelons ports `provides`) de même type ou de type plus spécifique.

Dans la littérature, les usages varient quant à la définition du terme «composant». Certains utilisent ce terme pour le type et d'autres pour les instances. Beaucoup l'utilisent pour les deux concepts sans distinction. Dans ce manuscrit, nous avons fait le choix de

parler de «type de composant» ou d'«instance de composant» pour éviter toute ambiguïté et de conserver le terme «composant» seul quand la distinction importe peu ou qu'elle peut être déduite du contexte.

Il existe de nombreux modèles de composants dédiés à diverses catégories d'applications. On peut par exemple citer le modèle de composants de CORBA [49] (CORBA *Component Model* – CCM) pour la construction d'applications d'entreprise. Les *Enterprise JavaBeans* [61] (EJB) se basent sur JAVA et sont particulièrement utilisés dans le cadre d'applications *web*. Le *Service Component Architecture* [50] (SCA) se focalise sur l'organisation d'applications à base de services *web*.

D'autres modèles sont issus du monde académique. C'est par exemple le cas de FRACTAL [18] initialement développé pour la conception de système d'exploitation et qui a été étendu pour l'adapter à une utilisation sur les grilles pour former le *Grid Component Model* [12] (GCM). On peut aussi citer la *Common Component Architecture* [4] (CCA) dédiée au calcul parallèle qui est présenté plus en détail en 2.3.1.

Dans certains modèles, les composants sont statiquement typés, leur interface étant alors définie à l'aide d'un langage spécifique. C'est par exemple le cas de CCM qui a étendu l'IDL CORBA dans ce but. Dans d'autres modèles, les composants sont dynamiquement typés et une API permet d'en spécifier les ports, c'est par exemple le cas de FRACTAL ou de CCA.

De la même manière, certains modèles comme CCM ou SCA proposent un langage dédié pour la description des assemblages de composant. On parle de langage de description d'assemblage (*Assembly Description Language* – ADL). D'autres s'appuient sur une instanciation dynamique des composants à l'aide d'une API dédiée comme FRACTAL ou CCA.

Suivant les modèles, les interactions entre composants peuvent s'appuyer sur divers mécanismes d'appel de méthode. Afin de permettre aux composants d'être développés dans des langages de programmation différents et de s'exécuter dans des espaces mémoires différents, la plupart des modèles s'appuient sur un mécanisme d'appel de méthode à distance. CCM s'appuie par exemple sur CORBA, les EJB sur les RMI JAVA et SCA propose une large gamme de mécanismes allant des services *web* aux appels de méthodes locaux. Le cas de CCA est particulier puisqu'il a été conçu pour être orthogonal au parallélisme et donc local. Cependant, il s'appuie sur l'outil BABEL qui permet d'effectuer des appels entre langages différents.

De nombreux modèles proposent un concept de port `uses` *multiple* qui peut être connecté à plusieurs ports `provides` compatibles pour permettre au composant d'effectuer l'appel sur le composant de son choix. C'est par exemple le cas de CCM ou de FRACTAL dans lequel on parle de «collection de ports».

Certains modèles proposent aussi un concept de hiérarchie qui consiste à permettre la mise en œuvre des composants par des assemblages d'autres composants. On parle de mises en œuvres composite des composants. Dans SCA, les composites permettent la réutilisation de parties d'assemblage en les encapsulant dans des composites qui délèguent leurs ports à des ports de même type des instances de composant qu'ils contiennent. Dans FRACTAL, les composites ont un rôle plus actif puisqu'ils intègrent un concept de membrane qui permet d'intercepter les appels de méthodes sur leur port avant de les retransmettre à leurs instances internes.

Il est fréquent qu'en plus des ports permettant de connecter les composants entre eux par des appels de méthodes, ces derniers exposent des attributs. Il s'agit de point de configuration acceptant une valeur qui peut être spécifiée dans l'assemblage pour modifier le comportement du composant.

Les divers modèles proposent aussi souvent divers services permettant aux composants de se concentrer sur le code métier. On peut par exemple noter la présence de services de persistance de données, de gestion de la sécurité ou bien de notions de transactions.

2.2.3 Les squelettes algorithmiques

Les squelettes algorithmiques sont des constructions qui ont pour objectif de décrire des structures de composition récurrentes dans un domaine [25]. Chaque squelette décrit une structure de composition spécifique en laissant les éléments ainsi composés être spécifiés par l'utilisateur. Lorsqu'une instance de squelette est créée, les éléments à utiliser pour ces points de paramétrisation doivent être spécifiés. Il peut s'agir d'éléments de code décrits par l'utilisateur ou d'autres instances de squelettes.

L'avantage de cette approche est qu'elle permet de découpler la mise en œuvre du code fonctionnel lui-même et des interactions efficaces entre les diverses parties de l'application. Plusieurs squelettes ont notamment été identifiés pour le cas du calcul scientifique à haute performance comme par exemple le *pipeline*, la ferme de tâches, le «*map reduce*», divers types d'itérations ou le «*divide and conquer*».

Ces différents squelettes correspondent à divers types de composition qui ont déjà été décrits. On peut par exemple mettre en relation la ferme de tâche qui correspond à l'exécution d'un même traitement sur un ensemble de données avec le paradigme maître/travailleurs. De la même manière, le *pipeline* qui propose d'appliquer de manière séquentielle un ensemble d'opérations sur des données peut être comparé au concept de flux de données.

Les modèles de squelettes algorithmiques sont parfois utilisables sous la forme de bibliothèques accessibles depuis des langages comme le C, le C++, le Java ou le Fortran. Dans ce cas, ils ne correspondent pas exactement à la définition que nous avons choisi pour les modèle de programmation par composition.

Il existe cependant des approches qui proposent de découper la description des éléments primitifs dans ces langages et leur composition à l'aide de squelettes avec un langage dédié. C'est par exemple le cas du *Pisa Parallel Programming Language* [9] (P^3L).

2.2.4 Analyse

Les modèles de programmation par composition permettent la séparation du développement des applications selon trois aspects :

1. le code métier est développé par des spécialistes du domaine au sein des éléments à composer ;
2. l'assemblage de ces éléments est exprimé par l'utilisateur en fonction de ses besoins à l'aide d'outils adaptés ; et
3. les aspects techniques sont gérés par le modèle lui même et peuvent ainsi être mis en œuvre par des spécialistes du matériel utilisé.

Ces caractéristiques offrent un cadre intéressant pour le développement d'applications de calcul scientifique.

Les abstractions offertes par les trois modèles qui ont été présentés sont caractéristiques d'objectifs différents. Les modèles de flux de travail sont adaptés à l'application de traitement similaires à une grande quantité de données. Leur principal avantage réside dans la possibilité de répartir de manière efficace ces traitements sur les cœurs d'exécution disponibles.

Les modèles de composants logiciels sont plus généralistes. Ils offrent la possibilité de construire des applications par la composition de codes qui possèdent un état interne et interagissent au cours de leur exécution. Il s'agit cependant de modèles qui imposent un couplage relativement élevé des assemblages de composants avec les ressources d'exécution.

Les modèles de squelettes algorithmiques se focalisent sur l'exécution efficace d'un ensemble de schémas de composition classiques. Ils sont par contre difficilement utilisables quand les applications ne sont pas exprimables à l'aide de l'ensemble limité des schémas proposés.

2.3 modèles de composants pour le calcul à haute performance

Les deux sections précédentes ont présenté des paradigmes de programmation utilisés pour permettre aux applications de calcul scientifique d'atteindre de hautes performances ainsi que des modèles de programmation par composition qui facilitent le développement d'applications à partir d'éléments d'origines diverses. Chaque modèle possède des points forts qui lui sont propres.

Cette section s'intéresse à la possibilité de combiner ces avantages. Il s'agit de permettre l'utilisation des paradigmes de programmation dédiés au calcul à haute performance dans un modèle de programmation par composition. Les modèles de flux de travail et de squelettes algorithmiques possèdent des propriétés qui les placent à la frontière entre modèles de composition et paradigme de programmation. Notre étude se focalise donc sur les modèles de composants dont les fonctionnalités sont plus orthogonales au problèmes des hautes performances.

Plusieurs approches peuvent être envisagées pour utiliser des paradigmes de programmation dédiés au calcul à haute performance dans les modèles de composants logiciels.

Une première approche consisterait à s'appuyer sur les bibliothèques qui mettent en œuvre les paradigmes dans les composants. Cependant, l'utilisation des moyens d'interaction entre flux d'exécution offerts par ces bibliothèques depuis les composants rompt leur aspect «boîte noire» en introduisant des interactions non spécifiées dans leur interface.

Une seconde approche consisterait à développer le code de mise en œuvre des divers paradigmes utilisés au sein même des composants en s'appuyant sur les ports du modèle pour les communications. Cette approche revient cependant à regrouper dans le composant le code métier et le code technique et elle lie le composant à une mise en œuvre du paradigme qui ne peut plus évoluer avec le matériel.

Une troisième approche consisterait à déporter ce code de support dans un composant dédié pour découpler son développement de celui du code métier et pour permettre son remplacement. Cette solution semble plus intéressante puisque qu'elle fixe la mise en œuvre du paradigme dans l'assemblage plutôt que dans les composants eux même. Cependant elle reste problématique pour la redistribution d'assemblages sous la forme de composites.

De ce fait, des modèles de composants qui intègrent en leur sein le support des paradigmes nécessaires ont été proposés. Cette approche permet de laisser les choix dépendants des ressources être fait lors du déploiement de l'assemblage et d'abstraire les aspects non fonctionnels pour le développeur de composants. Ce sont ces modèles qui sont présentés dans cette section.

2.3.1 Composants parallèles

Une première fonctionnalité dédiée au calcul à haute performance consiste en un support de composants parallèles qui permettent le développement d'applications SPMD. Les instances de composants parallèles sont formées d'un ensemble de processus qui s'exécutent en parallèle en communiquant par passage de message comme présenté en 2.1.2.

Le nombre de processus à utiliser pour chaque instance de composant parallèle et leur placement sont des choix fortement dépendant des ressources d'exécution. Similairement, le choix des algorithmes utilisés pour les opérations de communication collective entre processus ainsi que pour les appels de méthodes entre composants parallèles doivent être fait en prenant en compte ces ressources.

Mise en œuvre dans CCA Les composants parallèles constituent une fonctionnalité de base de CCA avec les concepts de *Single Component / Multiple Data* (SCMD) et de *Multiple Component / Multiple Data* (MCMD). La particularité d'une mise en œuvre parallèle de composant est qu'elle peut s'appuyer sur une bibliothèque de passage de messages telle que MPI ou PVM par exemple. Chaque instance d'un tel composant est formée d'un ensemble de

processus appelé *cohorte*. Les appels de méthodes entre composants parallèles sont décrits par des ports `uses` et `provides` classiques qui sont locaux à un processus. Puisque CCA est un modèle de composants dynamique, il est possible de décrire des ports différents dans chaque processus de la *cohorte*.

L'approche SCMD consiste à distribuer toutes les instances de composants parallèles sur le même ensemble de nœuds et à effectuer une connexion sur chaque nœud comme présenté sur la figure 2.1. Au contraire, l'approche MCMD consiste à distribuer chaque composant parallèle de calcul sur un ensemble distinct de nœuds et à utiliser un composant de couplage distribué sur la totalité des nœuds. Les composants de calculs sont tous connectés au composant de couplage comme présenté sur la figure figure 2.2.

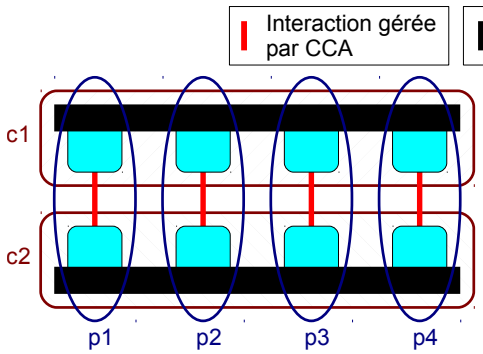

FIGURE 2.1 – Couplage SCMD de deux *cohortes* `c1` et `c2` sur quatre fils d'exécution `p1` à `p4`.

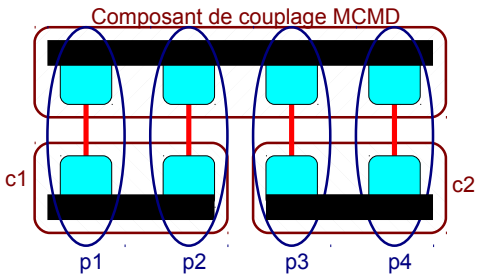

FIGURE 2.2 – Couplage MCMD de deux *cohortes* `c1` et `c2` sur deux fils d'exécution chacune.

À l'exécution, la gestion du parallélisme dans CCA est entièrement gérée par la bibliothèque choisie pour les communications entre processus. C'est cette bibliothèque qui est utilisée pour déployer les composants et pour les communications entre processus. Le code de support de CCA lui même n'est utilisé que localement à chaque processus pour le couplage des composants.

Cette approche est intéressante parce qu'elle simplifie le portage d'applications SPMD existantes vers CCA. Cependant, elle ne permet pas d'adapter une telle application pour utiliser une mise en œuvre destinée à d'autres types de ressources d'exécution. Le couplage de codes parallèles en s'appuyant sur CCA permet de s'abstraire de problèmes comme la compatibilité entre différents langages. Cependant si les données sont distribuées différemment entre les composants ainsi couplés, ils doivent effectuer la redistribution eux même. L'approche MCMD permet de déporter cette redistribution dans un composant dédié, mais ce composant doit être ré-écrit pour chaque nouveau couplage.

Mise en œuvre dans SCIRun2 SCIRun2 [64] est une mise en œuvre de CCA qui en étend le modèle pour y ajouter le support de l'appel parallèle de méthodes entre composants. Pour profiter de ce support, il est nécessaire de spécifier dans l'interface du composant quels ports `uses` ou `provides` sont parallèles. Les ports `uses` et `provides` parallèles doivent être connectés entre eux, il n'est pas possible de les connecter à des versions séquentielles.

Pour utiliser un port `uses` parallèle, tous les membres de la *cohorte* du composant avec le port `uses` doivent effectuer l'appel simultanément. Chacun de ces membres envoie une partie des paramètres, en fait uniquement la partie des données dont ils sont responsables. Chaque appel est retransmit au membre de la *cohorte* qui expose le port `provides` ayant le même identifiant modulo le nombre d'instances comme présenté dans la figure 2.3.

En tant qu'extension à CCA, SCIRun2 en possède les principaux avantages et inconvénients. Cette mise en œuvre simplifie le couplage de codes parallèles en permettant de décrire des interactions qui ne sont plus locales à un processus mais globales à un composant parallèle. La redistribution des données reste cependant du ressort des composants

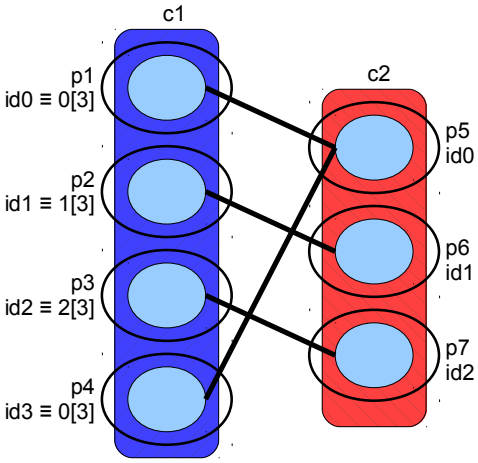

FIGURE 2.3 – Couplage de composants parallèles en $M \times N$ dans SCIRUN2.

utilisateurs, ce qui complexifie leur développement et limite leur réutilisation dans d'autres contextes.

Mise en œuvre dans GRIDCCM GRIDCCM [52] est une extension à CCM qui introduit le support des composants parallèles avec appel de méthode $M \times N$ entre composants. Une mise en œuvre parallèle de composant dans GRIDCCM est formée de deux parties : un descripteur de distribution des données et une mise en œuvre séquentielle. Le composant ainsi mis en œuvre expose automatiquement un attribut CCM utilisé pour spécifier le nombre d'instances de la mise en œuvre séquentielle à créer.

Comme dans CCA, les instances de la mise en œuvre séquentielle communiquent à l'aide d'une bibliothèque de passage de message. Les interactions entre composants parallèles se font par des port `uses` ou `provides` similaires à ceux des composants séquentiels. Le descripteur de distribution des données permet de dériver de l'interface du composant une interface interne utilisée par la mise en œuvre séquentielle comme présenté sur la figure 2.4. Pour utiliser un port `uses` (réciproquement un port `provides`), chaque fil d'exécution du composant exécute (réciproquement reçoit) un appel de méthode avec en paramètre la partie des données qu'il gère.

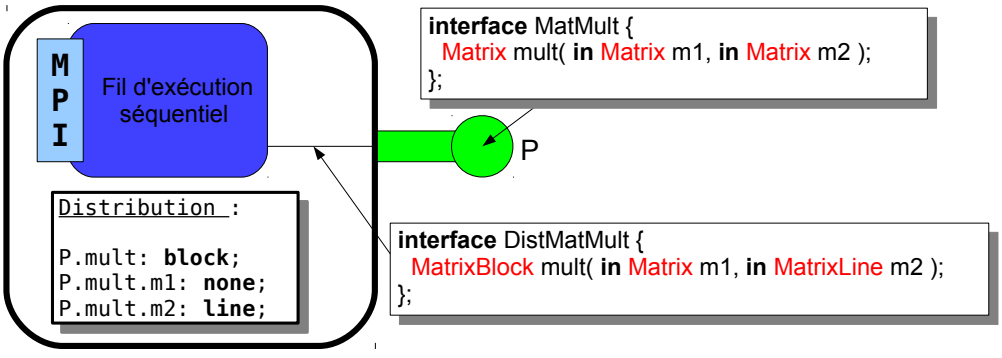

FIGURE 2.4 – Mise en œuvre parallèle d'un composant dans GRIDCCM.

Quand un composant parallèle est connecté à un composant séquentiel ou à un autre composant parallèle utilisant une distribution de données différente, il est nécessaire de redistribuer les paramètres et la valeur de retour des méthodes exposées par les ports. Cette redistribution est effectuée par du code inséré automatiquement à la compilation du composant par un outil spécifique à GRIDCCM. L'instanciation de ce code de redistribution

est effectuée de manière transparente lors de la connexion en s'appuyant sur une interface d'introspection qui permet de découvrir si le composant auquel on se connecte est parallèle et s'il l'est, quelle distribution de données il utilise.

Par rapport à CCA, GridCCM offre une interface de plus haut niveau qui conserve les propriétés des ports `uses` et `provides` pour les composants parallèles. En supportant la redistribution des données de manière transparente pour l'utilisateur, celui-ci n'a pas à se préoccuper de ces aspects. L'adaptation du comportement du composant parallèle aux ressources d'exécution est cependant limitée aux algorithmes présents dans la bibliothèque de passage de messages et dans son code de redistribution.

Une autre difficulté peut être liée à l'utilisation de deux bibliothèques différentes pour les communications entre processus distants : une bibliothèque de passage de message et Corba. Cet aspect a rendu nécessaire le développement d'un outil permettant d'utiliser plusieurs bibliothèques de communication au sein d'une même application : PadicoTM [26].

Mise en œuvre par des composants répliquants Le concept de répliquants [BP07] a été proposé comme une autre extension à CCM. Cette extension est destinée à permettre la mise en œuvre de composants parallèles en tant que composite contenant un ensemble d'instances du composant séquentiel. Elle introduit deux concepts dans ce but : les mises en œuvre répliquantes de composant et les connexions `AnyToAny`. Ces extensions sont utilisées pour la mise en œuvre de composants parallèles mais aussi pour faciliter leurs interactions avec la mise en œuvre d'un composant de communication collective.

Une mise en œuvre répliquante de composant met en œuvre un type de composant à l'aide d'un autre composant qui est répliqué. Elle est formée de quatre parties : le type du composant répliqué, un ensemble de connexions `AnyToAny` internes et une mise en correspondance des autres ports du composant répliqué avec ceux exposés par la mise en œuvre répliquante. Le degré de réplication peut être spécifié soit explicitement soit lié aux ressources d'exécution.

Une connexion `AnyToAny` regroupe plusieurs ports `AnyToAny` entre eux. Un port `AnyToAny` est typé par une interface objet, comme les port `uses` ou `provides`. Il se traduit en pratique par deux ports du même type, un port `uses` *multiple* et un port `provides`. Une connexion `AnyToAny` se traduit par une connexion `uses/provides` entre chaque port `uses` *multiple* et chaque port `provides` participant. Ces ports permettent d'offrir des interactions relativement similaires aux interactions de passage de message point à point.

Le composant de communication collective est utilisé pour permettre aux composants dans une mise en œuvre répliquante de communiquer par ces opérations de passage de message de plus haut niveau que celles point à point. Il ne s'agit pas d'une extension du modèle de composants à proprement parler, mais d'un composant exposant un port `provides` avec une interface dédiée et dont la mise en œuvre est répliquante.

Au déploiement, lorsque les ressources d'exécution sont connues, une opération de transformation de l'assemblage est appliquée dont un exemple est présenté sur la figure 2.5. Les composants répliquants sont remplacés par un ensemble d'instances du composant répliqué et les connexions `AnyToAny` sont remplacées par leur équivalent en terme de connexions entre ports `uses` *multiple* et `provides`. Le résultat est un assemblage CCM qui ne comporte plus de spécificités et peut être déployé par les outils classiques.

En permettant aux composants parallèles d'être formés d'un ensemble d'instance de composant séquentiels, cette approche permet de réduire la granularité à laquelle les composants peuvent être utilisés par rapport à GridCCM. Les interactions efficaces entre composant parallèles ne sont cependant pas évoquées pour cette mise en œuvre qui nécessite donc comme pour CCA que les composants les gèrent entièrement. De plus, en nécessitant le passage par des ports du modèle de composants pour les communications entre processus du composant parallèle, le portage d'applications existantes peut être complexe. A l'inverse, ce choix a l'avantage de permettre le remplacement du code de mise en œuvre des communications entre processus de manière simple pour s'adapter à de nouvelles ressources d'exécution.

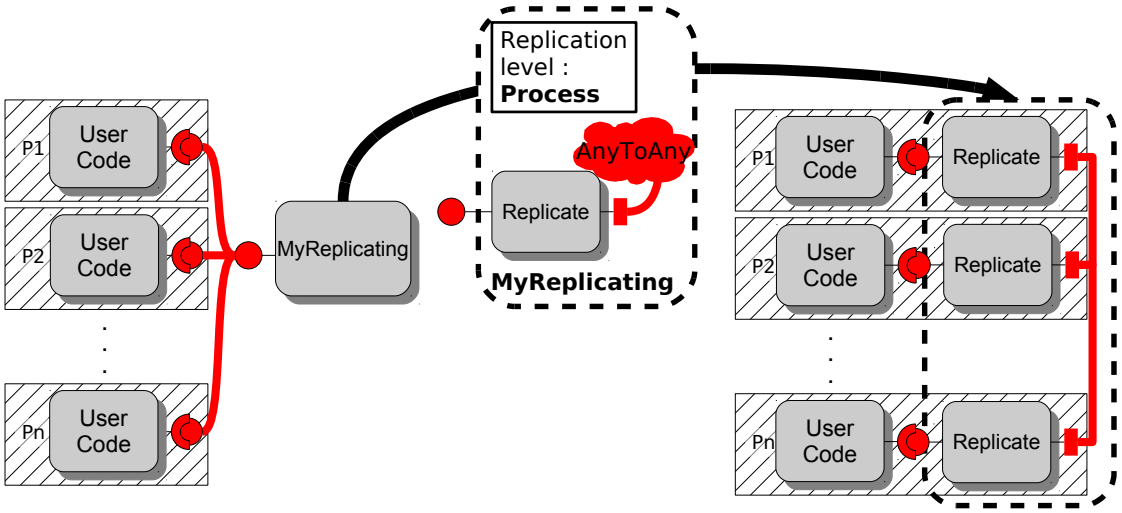

FIGURE 2.5 – Exemple de transformations d'un assemblage comportant un composant répliquant.

2.3.2 Partage de données entre composants

Une seconde fonctionnalité dédiée au calcul à haute performance consiste en un support du partage de données par les composants pour gérer le paradigme de mémoire partagée. Il s'agit de permettre à un ensemble d'instances de composants de travailler sur une même donnée à laquelle ils accèdent au travers de ports dédiés.

La mise en œuvre efficace d'un tel partage dépend fortement du placement des instances de composants y participant sur les ressources d'exécution.

Mise en œuvre Le support du partage de données entre composants a été proposé comme une extension à CCM [6] et a ensuite été appliqué à CCA [7]. Cette extension introduit deux nouveaux types de ports : `share` et `access` qui permettent respectivement de partager une donnée et d'accéder à une donnée ainsi partagée. Chaque port `access` doit être connecté à un unique port `share` dans l'assemblage de composants.

Le fait pour un type de composant d'exposer un port `share` ou `access` donne accès à ses mises en œuvres à une interface interne qu'elles peuvent utiliser comme dans le cas des ports `uses`. Dans le cas des ports `access` et `share`, le type de l'interface accessible est fixé. Il s'agit respectivement des interfaces `AccessPort` et `SharePort` présentées dans les figures 2.6 et 2.7. Ces interfaces permettent d'obtenir une adresse locale pour la donnée et offrent des primitives de synchronisation. En plus l'interface `SharePort` permet de définir la taille ainsi que la valeur de la donnée partagée.

```
interface AccessPort {
  void* get_pointer();
  long get_size();
  // primitives de synchronisation
  void  acquire();
  void  acquire_read();
  void  release();
}
```

FIGURE 2.6 – Interface d'accès aux données aux travers d'un port `access`.

```
interface SharePort : AccessPort {
  void set_pointer( void* data,
    long size );
}
```

FIGURE 2.7 – Interface de partage de données au travers d'un port `share`.

Les composants exposant de tels ports doivent être compilés à l'aide d'un outil particulier. Cet outil insère dans le composant le code d'une bibliothèque qui met en œuvre le partage de donnée. Il peut s'agir d'un simple accès à la mémoire si les composants sont dans le même processus, d'un segment de mémoire partagée s'ils sont sur le même nœud, d'une mémoire partagée distribuée sur une grappe de calcul ou d'une solution pair à pair telle que JuxMem sur une grille de calcul. Il ajoute aussi à l'interface du composant de nouveaux ports utilisés par cette mise en œuvre pour échanger ses données de configuration. La connexion dans l'assemblage d'un port `access` à un port `share` se traduit à l'exécution par la connexion de ces ports additionnels.

En permettant de projeter dans la mémoire du composant une donnée partagée, les ports `share` et `access` offrent une interface simple afin d'utiliser ce paradigme. Le choix de la mise en œuvre à utiliser doit cependant être fait lors de la compilation du composant ce qui complique le changement de mise en œuvre en cas de changement important des ressources d'exécution utilisées.

2.3.3 Flux de de travail dans les modèles de composants

L'ajout du support des flux de travail dans un modèle de composants permet d'offrir un modèle où les compositions sont à la fois spatiales et temporelles. Il s'agit d'introduire le concept de *tâche* dans le modèle et de gérer leur cycle de vie et interactions spécifiques.

Des tâches sans état permettent de laisser de nombreux paramètres être déterminés lorsque les ressources de calcul sont connues comme par exemple le placement de leurs instances, leur réutilisation ou leur réplication.

Mises en œuvre dans STCM La gestion des flux de travail a été proposée dans STCM [15] qui étend GCM avec des concepts de flux de travail issus d'AGWL. STCM introduit les concept de *tâche* et de *composant-tâche* ainsi que de nouvelles catégories de ports, les ports d'entrée et de sortie. Il offre aussi un langage de description d'assemblage de *composants-tâches* qui combine les dimensions spatiale et temporelle.

Les ports d'entrée et de sortie sont utilisés pour décrire le transfert asynchrone de données entre *tâches* dans STCM. Ils sont typés par le type de la donnée transmise. Puisque Fractal—et donc GCM— sont des modèles de composants dynamiques, ces nouvelles catégories de ports sont offertes en étendant l'API utilisée pour décrire un type de composant. Ils sont mis en relation avec un *getter* ou un *setter* de la classe de mise en œuvre du composant.

Afin de pouvoir être utilisé comme *tâche*, un *composant-tâche* doit exposer un port `provides` d'un type spécifique : `TaskControler` qui contient une unique méthode `use` sans paramètre ni résultat. Cette méthode est appelée par le moteur d'exécution de flux de travail pour exécuter la *tâche* associée au *composant-tâche*. L'automate d'états fini utilisé pour décrire le cycle de vie des composants dans GCM est étendu dans STCM avec un nouvel état `lancé` (`running`) qui correspond à un *composant-tâche* en cours d'exécution de la *tâche* qui lui est associée comme présenté sur la figure 2.8.

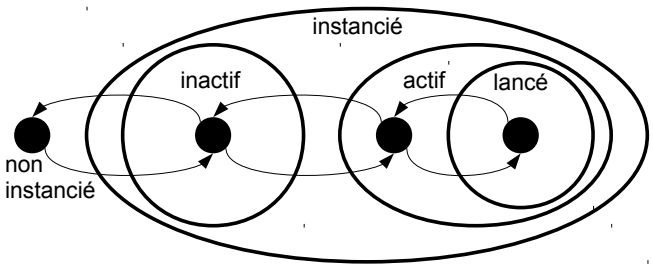

Figure 2.8 – Automate décrivant le cycle de vie d'un *composant-tâche* STCM.

L'exécution d'un application STCM s'appuie sur un moteur d'exécution de flux de travail

qui instancie les composants qui doivent l'être et lance les *tâches* en utilisant l'API de GCM étendue dans ce but. Les *composants-tâches* sont instanciés

- soit lorsque la *tâche* qu'ils supportent est lancée pour la première fois ;
- soit lorsqu'une instance de composant auquel ils sont connectés par un port `provides` est instanciée.

Leur destruction suit la logique inverse.

En permettant de décrire des *tâches* dans un modèle de composants, STCM permet d'abstraire pour l'utilisateur la gestion des interactions temporelles entre composants. Le moteur d'exécution de flux de travail est responsable de la gestion du cycle de vie des *composants-tâches* qui peut être adapté aux ressources d'exécution disponibles.

2.3.4 Squelettes algorithmiques dans les modèles de composants

Les squelettes algorithmiques peuvent être considérés dans les modèles de composants comme des patrons de composites au comportement paramétrable. Les squelettes possèdent un ensemble de paramètres pour spécifier leur comportement ainsi qu'un second ensemble de paramètres pour les adapter aux ressources d'exécutions.

Nous nous intéresserons ici en particulier à l'exemple de la ferme de tâches utilisée pour mettre en œuvre le paradigme maître/travailleurs. Le nombre d'instances du travailleur à déployer, ainsi que la mise en œuvre du code de répartition de charge à utiliser sont deux aspects fortement dépendants des ressources d'exécution.

Mise en œuvre de la ferme de tâches Une première mise en œuvre limitée à la ferme de tâches a été proposée comme extension à CCM et Fractal qui a ensuite été appliquée à CCA [16]. Cette extension introduit le concept de *collection* qui permet de décrire une ferme de tâches dans un assemblage de composants.

Une *collection* est définie par deux parties : un type de composant à répliquer (le travailleur) et une mise en correspondance des ports du travailleur avec ceux de la *collection*. Il existe deux stratégies possibles pour mettre en correspondance un port `uses` du travailleur avec un port de la *collection*. Il peut être mis en correspondance avec un port `provides` ou avec un port `provides` *multiple*[2]. Les ports `uses` du composant répliqué sont simplement mis en correspondance avec un port `uses` de la collection.

Le déploiement d'un assemblage comportant des *collections* s'appuie sur une phase de transformation pendant laquelle les ressources d'exécutions disponibles sont prises en compte. Deux choix sont fait à cette étape : le nombre d'instances du travailleur à créer et le code de répartition de charge à utiliser avec ses éventuels paramètres. Le résultat est un assemblage qui ne comporte plus de spécificités et peut être déployé par les outils classiques.

Bien qu'elle soit limitée à la ferme de tâche, cette mise en œuvre est intéressante car elle permet d'utiliser les fermes de tâches dans le modèle de composants sans modification particulière des composants. La simplicité d'utilisation est maximisée puisque les interactions avec la ferme reposent sur l'utilisation de ports `uses` et `provides` classiques. Finalement en laissant la transformation être appliquée au déploiement, l'assemblage peut être adapté aux ressources d'exécutions aussi bien en choisissant le nombre de travailleurs qu'en changeant le code d'équilibrage de charge utilisé.

Mises en œuvre dans STKM Une mise en œuvre des squelettes algorithmiques plus générale a été proposée dans STKM [3], une extension à STCM. STKM introduit des squelettes qui peuvent être utilisés de manière similaire aux composants dans l'assemblage. De nouvelles catégories de ports sont aussi introduites pour définir les interactions entre squelettes : les ports de flot de données en entrée et en sortie.

2. Il s'agit d'un concept similaire aux collections de ports de Fractal à ne pas confondre avec le concept de collection d'instance de composants discuté ici.

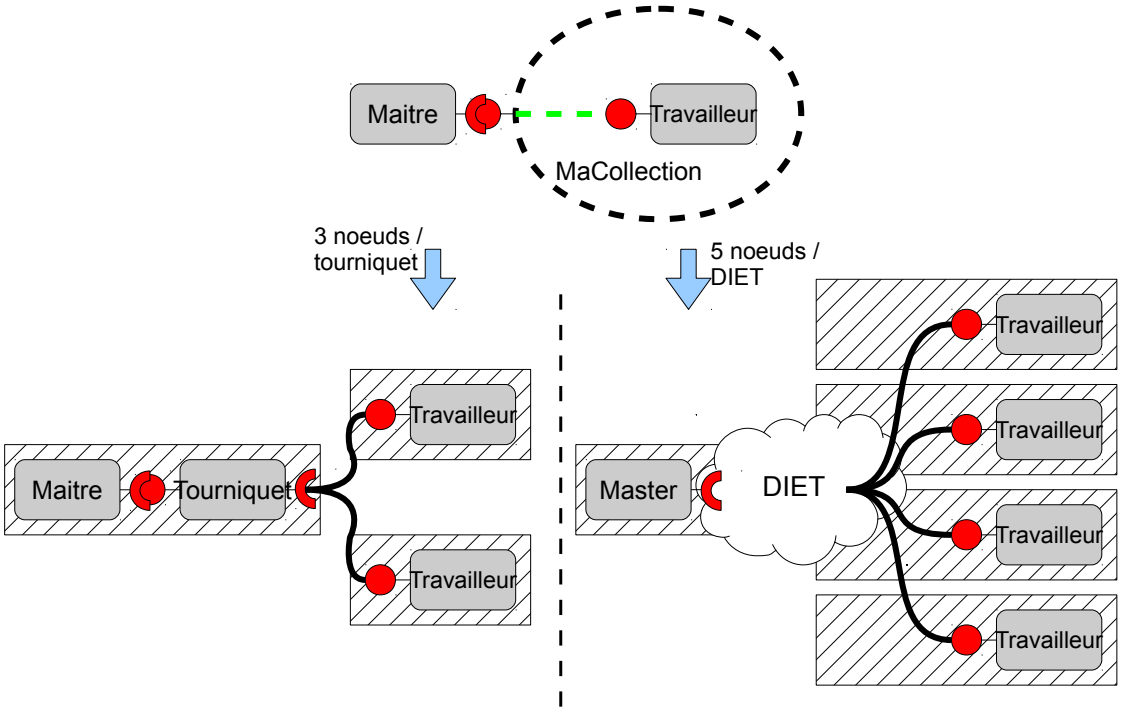

FIGURE 2.9 – Deux transformations possibles d'une collection au déploiement, 3 instances avec répartition de charge tourniquet (*round-robin*) et 5 instances avec répartition de charge utilisant DIET.

Les squelettes dans STKM offrent une interface externe définie par un ensemble de ports à la manière des composants et des interfaces internes ou points de paramétrisation. Chaque point de paramétrisation attend un type de composant et expose des ports internes comme présenté sur la figure 2.10. Lors de l'instanciation d'un squelette, chacun de ses points de paramétrisation doit être rempli. Pour cela le type de composant à utiliser doit être spécifié et ses ports mis en relations avec ceux du point de paramétrisation. Les ports d'un composant utilisé pour remplir un point de paramétrisation peuvent aussi être connectés à des composants extérieurs au squelette, comme présenté sur la figure 2.11.

Lors du déploiement d'un assemblage comportant des squelettes, un outil dédié est utilisé qui déploie à la place de chaque instance de squelette un composite contenant la mise en œuvre du squelette. Le contenu de ce composite peut varier en fonction des ressources d'exécution disponibles. Chaque entité utilisée pour remplir un point de paramétrisation peut être instanciée zéro, une ou plusieurs fois. Il est aussi possible d'insérer dans cet assemblage résultat une instance de composant de gestion chargé de maintenir des propriétés d'auto adaptation. Un exemple d'assemblage résultant de l'instanciation du squelette présenté en figure 2.11 est présenté figure 2.12.

En étendant STCM, STKM offre un comportement des squelettes algorithmiques basé sur des flux de données. Ce comportement est plus proche de celui des mises en œuvre classiques des squelettes algorithmiques que la mise en œuvre présentée précédemment. De ce fait, STKM hérite aussi des avantages des modèles de flux de travail et en particulier de STCM pour la gestion du cycle de vie des *composants-tâches*. Le résultat est un modèle qui abstrait de nombreux points pour l'utilisateur et qui permet une adaptation aux ressources en choisissant la mise en œuvre de chaque squelette la mieux adaptée lorsque les ressources d'exécution sont connues.

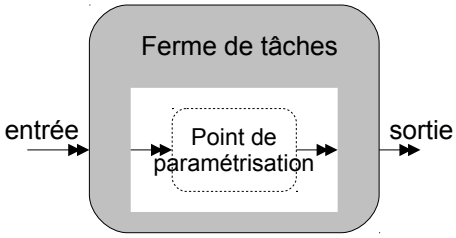

FIGURE 2.10 – Exemple de squelette «ferme de tâches» qui expose deux ports de flot de données, un en entrée et un en sortie et un point de paramétrisation avec.

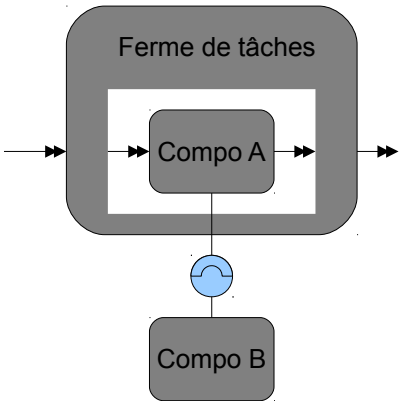

FIGURE 2.11 – Une instance du squelette «ferme de tâches» dont le point de paramétrisation est rempli par un composant connecté à une instance de composant externe au squelette.

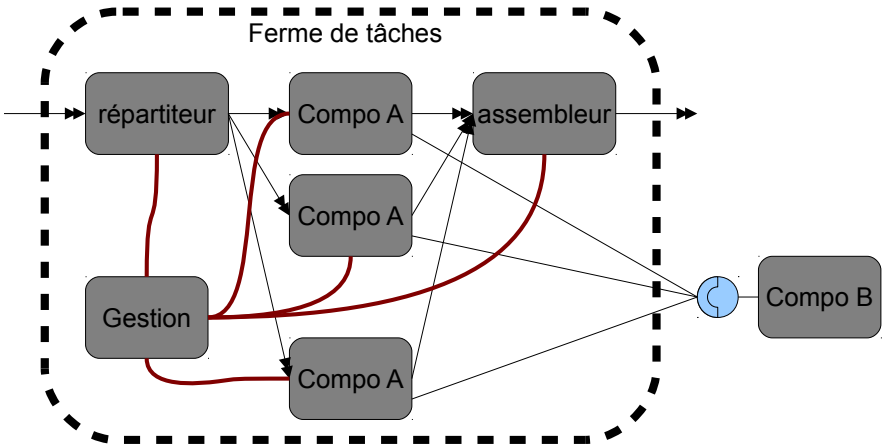

FIGURE 2.12 – Une ferme de tâches STKM déployée. Les lignes courbes représentent les liens de surveillance par le composant de gestion.

2.3.5 Analyse

L'ajout de fonctionnalités dédiées au calcul à haute performance aux modèles de composants leur permet d'offrir une alternative intéressante pour le développement des applications de calcul scientifique. Ils permettent de s'appuyer sur les paradigmes classiques pour la programmation d'applications parallèles et distribuées tout en permettant une analyse aisée de l'architecture des applications et en facilitant la réutilisation de code existant.

Les fonctionnalités qui ont été introduites pour supporter ces paradigmes sont diverses. Elles peuvent être classifiées selon deux axes :

1. certaines introduisent de nouvelles catégories d'interactions entre les composants alors que les autres introduisent de nouvelles manière pour mettre en œuvre les composants ;
2. certaines fonctionnalités sont dynamiques, elles influencent le comportement de l'assemblage au cours du temps ; les autres sont statiques, elles n'ont d'influence que sur l'assemblage initial.

Les nouvelles catégories d'interactions entre les composants et de mise en œuvre de composants introduites par les diverses fonctionnalités présentées peuvent être résumées

par le tableau suivant. Dans ce tableau on a représenté une ligne pour la ferme de tâches, chaque squelette algorithmique nécessiterait une ligne, mais la ferme de tâches est le seul qui ait été étudié.

	interactions	mises en œuvre
SPMD	appels de méthodes $M \times N$ entre composants parallèles, passage de messages, communication collective	composants parallèles
partage de données	partage de données	aucun
flux de travail et flux de données	flux de données, flux de contrôle	*composant-tâche*
ferme de tâches	appel de méthode avec équilibrage de charge	ferme de tâches

À l'exception du partage de données, chaque fonctionnalité repose sur l'introduction de plusieurs concepts. Pour chacune de ces fonctionnalité, des concepts appartenant aux deux catégories sont introduits.

Concernant l'aspect statique ou dynamique des concepts introduits par les fonctionnalités, la majorité sont statiques. Les flux de données et flux de travail forment la fonctionnalité la plus clairement dynamique puisque son principal impact est sur le cycle de vie des *composants-tâches*. Il faut cependant noter que les squelettes algorithmiques tels que mis en œuvre dans STKM ont aussi un aspect dynamique pour la gestion de l'auto adaptation.

2.4 Conclusion

Ce chapitre a présenté des paradigmes adaptés au développement d'applications à destination des ressources parallèles et distribuées qui permettent d'obtenir de hautes performances. Il a aussi présenté des modèles de programmation par composition adaptés au développement d'application constituées de codes d'origine diverses. Finalement, des modèles qui combinent ces deux aspects et qui sont donc particulièrement adaptés au développement d'applications de calcul scientifique ont été présentés.

Les approches qui ont été adoptées pour combiner ces deux aspects consistent à introduire de nouveaux concepts dans les modèles de composants logiciels afin de décrire à un plus haut niveau d'abstraction certains aspects de l'application. Cette augmentation du niveau d'abstraction simplifie le travail des développeurs de composants qui n'ont alors plus à déterminer comment mettre en œuvre ces aspects, mais seulement quels aspects utiliser. Elle permet aussi de simplifier l'adaptation des composants à de nouvelles ressources d'exécutions en découplant la mise en œuvre de ces aspects de celle du composant. Finalement, elle permet une meilleure lisibilité de l'architecture de l'application en ôtant des aspects techniques des assemblages de composants.

Les travaux actuels sont cependant limité au support d'un ensemble restreint de paradigmes. Il n'existe pas non plus de modèle permettant d'utiliser l'ensemble des paradigmes au sein d'une même application. Le chapitre suivant étudie donc la possibilité de proposer une approche pour faciliter l'introduction de nouvelles fonctionnalités au sein des modèles de composants logiciels.

CHAPITRE 3

Notre approche pour l'extension de modèles de composants

Le chapitre précédent a présenté des fonctionnalités intégrées aux modèles de composants pour les adapter au cas d'utilisation du calcul à haute performance. Ce chapitre présente et analyse les différentes approches qui ont été utilisées pour mettre en œuvre ces fonctionnalités, notamment en étendant des modèles de composant existant. Les limitations de ces approches nous conduit à en proposer une nouvelle qui repose sur un modèle conçu pour permettre la définition de nouvelles fonctionnalités sans nécessiter de modification du modèle lui-même.

La section 3.1 présente les diverses approches qui ont été utilisées pour étendre les modèles de composants avec des fonctionnalités dédiées au parallélisme. La section 3.2 propose ensuite une classification de ces approches selon trois critères pour former des catégories qui sont sont analysées dans la section 3.3. La section 3.4 présente l'approche que nous proposons pour définir un modèle de composant extensible. Finalement, la section 3.5 conclue le chapitre.

3.1 Présentation de l'existant

Parmi les propositions de fonctionnalités dédiées au calcul à haute performance présentées dans le chapitre 2 certaines spécifient un nouveau modèle de composants et s'attardent

peu sur leur mise en œuvre effective. Pour celles qui proposent des mises en œuvres, diverses approches ont été utilisées que nous présentons dans cette section.

3.1.1 Approche par extension du code de l'exécutif

Une première approche a été utilisée pour la mise en œuvre de fonctionnalités en étendant CCA ou FRACTAL. Ces deux modèles appartiennent à la catégorie des modèles dynamiques, c'est à dire que les composants et leurs assemblages y sont décrits par l'utilisation d'une API définie dans la spécification du modèle. L'approche classique pour étendre un tel modèle consiste a étendre cette API pour supporter la description de concepts additionnels.

Dans cette approche, le code de mise en œuvre des fonctionnalités additionnelles est donc distribué avec le code de support à l'exécution du modèle (l'exécutif) qui met en œuvre l'API. Les divers choix d'optimisation sont effectués à l'exécution —lors de l'appel des fonctions additionnelles— puisqu'il n'est pas possible de prévoir quelles seront les fonctionnalités utilisées avant cette étape. Ces choix peuvent être soit effectués automatiquement par l'exécutif soit être fait manuellement et passés sous la forme de paramètres aux fonctions additionnelles de l'API.

3.1.2 Approche par compilation dans les composants

Une seconde approche est celle qui a été utilisée dans la mise en œuvre de GRIDCCM pour supporter les composants parallèle ainsi que la redistribution de données dans les appels de méthode $M \times N$. Cette approche s'appuie sur une description des composants étendue par rapport à celle de CCM. Une chaîne de compilation dédiée présentée figure 3.1 utilise cette description pour insérer dans la mise en œuvre du composant du code additionnel qui supporte la redistribution. Le composant résultant offre une interface comportant des parties automatiquement générées. Un attribut est utilisée pour configurer le nombre d'instance du composant parallèle. Une interface dédiée est utilisée pour les interactions $M \times N$ entre composants parallèles. A l'exécution, le comportement des composants GRIDCCM ne suit pas exactement le comportement habituel des composants CCM puisque la sémantique de l'instanciation et de la connexion sont modifiée. L'instanciation d'un seul composant GRIDCCM donne par exemple lieu à la création de composants CCM dans plusieurs processus d'exécution. De même la connexion de deux composants parallèles donne lieu à plusieurs connexions entre composants CCM.

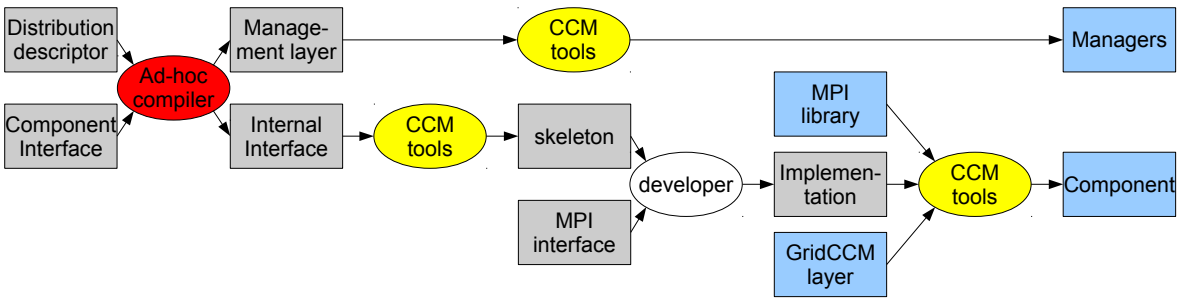

FIGURE 3.1 – Chaîne de compilation dédiée de GRIDCCM

Dans cette approche, les choix d'optimisations sont fait à plusieurs niveaux. Un premier choix concerne la ou les mises en œuvres de la redistribution à inclure dans les composants, il est effectué par la chaîne de compilation en s'appuyant sur les informations fournies par l'utilisateur. Un deuxième choix concerne de degré de réplication des composants parallèles, il est effectué manuellement lors du déploiement de l'application en définissant la valeur d'un attribut du composant. Finalement, le dernier choix concerne la mise en œuvre de la redistribution à utiliser parmi celles disponibles dans le composant, il est effectué à

l'exécution par le code inséré en s'appuyant sur de l'introspection pour étudier l'assemblage dans lequel l'instance de composant est utilisée.

3.1.3 Approche par compilation et utilisation de bibliothèque additionnelle

Une troisième approche relativement similaire à celle utilisée pour mettre en œuvre GridCCM, est celle qui a été utilisée dans la mise en œuvre des données partagées comme extension à CCM. Cette approche s'appuie sur une chaîne de compilation dédiée présentée figure 3.2 pour interpréter la définition étendue du composant et pour insérer du code dans le composant. Cependant, à l'inverse du cas de GridCCM, ce code n'est pas auto suffisant, il s'agit d'un code d'adaptation pour une bibliothèque qui doit être présente à l'exécution, telle que JuxMem par exemple. L'interface exposée par le composant résultant de cette compilation comporte de plus des ports spécifiques utilisées pour l'échange d'informations nécessaire à la connexion au travers de la bibliothèque utilisée. Un outil spécifique est donc nécessaire pour traiter les assemblage de composants utilisant les ports de partage de donnée qui traduit la connexion de ces ports logiques en connexion des ports effectivement présents.

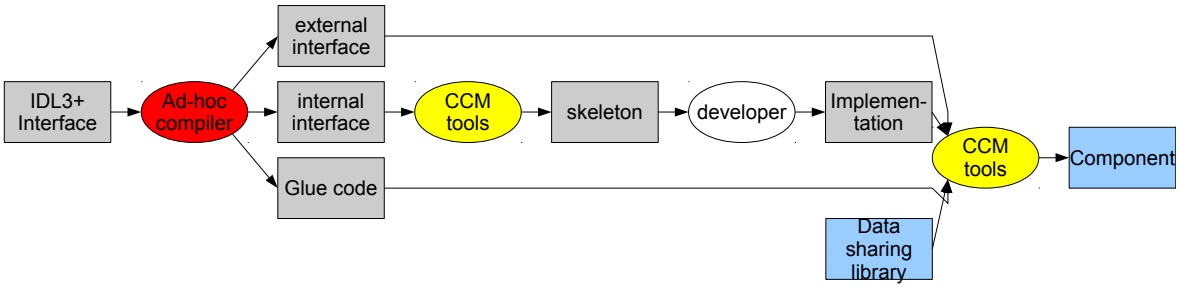

Figure 3.2 – Chaîne de compilation dédiée pour la mise en œuvre du partage de données dans CCM

Dans cette approche, comme dans celle utilisée pour mettre en œuvre GridCCM, les choix d'optimisations ont lieux à plusieurs niveaux. Un premier choix fait lors de la compilation des composants concerne la bibliothèque à utiliser, ce choix conditionne l'interface effectivement exposée par le composant. Ce choix est *a priori* fait manuellement lors de la compilation du composant. Les autres choix possibles dépendent de la bibliothèque utilisée et sont donc fait par celle-ci à l'exécution.

3.1.4 Approche par transformation de l'assemblage à l'exécution

L'approche utilisée pour mettre en œuvre les fermes de tâches dans CCM consiste en une transformation de l'assemblage comportant des concepts spécifiques à la fonctionnalité (les *collections*) en un assemblage CCM classique lors du déploiement. Chaque *collection* est remplacée par un ensemble d'instance d'un autre type de composant dont le nombre exact est défini par l'utilisateur qui effectue le déploiement. L'utilisation d'un outil pour effectuer ce choix automatiquement est aussi envisageable. De plus chaque connexion qui implique le port d'une *collection* est remplacée par des connexions vers les ports de ces composants. A cette étape, un ou plusieurs composants mettant en œuvre la répartition de charge peuvent être insérés.

Cette approche ne nécessite pas de modification des composants décrits par l'utilisateur, le code qui met en œuvre les fonctionnalités est inclus dans des composants additionnels. Ces composants additionnels sont automatiquement insérés dans l'assemblage lors de son déploiement par un outil dédié qui prend en compte les ressources d'exécution disponibles

pour effectuer les différents choix, choix de la mise en œuvre à utiliser et de ses éventuels paramètres.

3.1.5 Approche par programmation sous la forme de composants

L'approche utilisée pour mettre en œuvre les opérations de communication collective dans CCM consiste comme pour les fermes de tâches en une transformation de l'assemblage au déploiement. Une différence importante vient du fait que dans ce cas, ce n'est pas directement le concept mis en œuvre (les opérations de communication collective) qui est transformé. À la place, des concepts de plus bas niveau (composants répliquants et connexions `AnyToAny`) – qui permettent la mise en œuvre des opérations de communication collective – sont transformés. Chaque connexion `AnyToAny` est simplement remplacée par un ensemble de connexions `uses/provides` entre chaque paire de ports qui y participent. Chaque instance de composant répliquant est remplacé similairement aux *collections* par un ensemble d'instances d'un autre type de composant. Dans ce cas, le nombre d'instance à créer dépend des ressources sur lesquelles le composant est déployé et d'une indication fournie par le développeur du composant (autant que de processus, que de nœuds ou que de grappes de calcul par exemple). Dans cette approche, les composants peuvent posséder plusieurs mises en œuvres dédiées à divers types de ressources d'exécution. Le choix de la mise en œuvre à utiliser est aussi fait lors de cette phase de transformation en fonction des indications offertes par le développeur du composant.

3.2 Classification

Les approches précédemment présentées peuvent être classifiées selon plusieurs critères. Nous nous intéressons ici à une classification selon trois critères principaux :

1. l'emplacement où le code qui met en œuvre la fonctionnalité est inséré ;
2. la ou les étapes auxquelles les choix d'optimisation sont effectués ; et
3. le support offert par la mise en œuvre pour automatiser ces choix.

Localité du code Un premier critère concerne la localité où le code de mise en œuvre de la fonctionnalité est inséré. Trois possibilités existent :

1. ce code peut être directement inséré dans les composants qui utilisent la fonctionnalité ;
2. il peut être inséré sous la forme de nouvelles instances de composant dans l'assemblage ; ou
3. il peut être distribué avec l'exécutif du modèle de composants.

Les catégories auxquelles les approches présentées précédemment appartiennent sont résumés dans le tableau 3.1. Le cas de l'approche utilisée pour la mise en œuvre du partage de donnée est un peu particulier puisque dans cette approche du code est inséré dans les composants mais il n'est utilisée que pour interfacer le composant avec une bibliothèque externe. Dans la mesure où cette bibliothèque doit être présente afin que les composants fonctionnent correctement sans être spécifiée explicitement dans leur interface, nous la considéreront comme faisant partie du code de support du modèle et nous classons cette approche dans la troisième catégorie.

Temporalité des choix Nous avons vu que les fonctionnalités dédiées au calcul à haute performance sont caractérisées par le fait que leur mise en œuvre efficace dépend d'un certain nombre de choix qui doivent être faits en fonction des ressources d'exécution disponibles. Un second critère de classification consiste à distinguer le moment où ces choix sont fait. Ces choix peuvent être fait :

	dans les composants	dans l'assemblage	dans l'exécutif
Fractal et CCA			X
GridCCM	X		
partage de données			X
fermes de tâches		X	
communication collective		X	

Table 3.1 – Localité du code de support dans les diverses approches présentées

1. à la compilation des composants ;
2. lors du déploiement de l'assemblage de composants ou ;
3. au cours de l'exécution de l'application.

Les catégories auxquelles les approches présentées précédemment appartiennent sont résumés dans le tableau 3.2. Il faut noter que dans plusieurs approches, des choix sont fait à plusieurs étapes. Il s'agit de situations où des choix généraux sont d'abord fait pour être raffinés par la suite.

	compilation	déploiement	exécution
Fractal et CCA			X
GridCCM	X	X	X
partage de données	X		X
fermes de tâches		X	
communication collective		X	

Table 3.2 – Temporalité des choix d'optimisation des mises en œuvre dans les diverses approches présentées

On peut noter que dans les deux approches qui reposent sur une modification de l'assemblage de composant, aucun choix n'est fait à la compilation. Cet état de fait est dû à ce que lors de la compilation des composants, les assemblages dans lesquels ils seront utilisés ne sont pas encore connus ce qui rend ces deux catégories incompatibles entre elles.

Responsabilité des choix Un troisième critère qui permet de classifier les différentes approches consiste à identifier si le code de mise en œuvre des fonctionnalités est aussi responsable des choix d'optimisations ou si ces choix sont fait par une entité externe. Un cas intermédiaire existe aussi où la mise en œuvre exporte les informations utilisées pour effectuer les choix sans en être directement responsable. Les différentes possibilités sont donc les suivantes :

1. les choix sont directement effectués par le code de mise en œuvre de la fonctionnalité ;
2. le code de mise en œuvre exporte des informations qui sont utilisées pour effectuer les choix ; ou
3. les choix sont fait de manière externe sans aucun recours au code de mise en œuvre de la fonctionnalité.

Les catégories auxquelles les approches présentées précédemment appartiennent sont résumés dans le tableau 3.3. Il faut noter que comme pour le critère précédent, il existe des approches qui appartiennent à plusieurs catégories. Il s'agit à nouveau de situation où les choix sont décomposés en sous-choix pouvant être effectués selon des stratégies différentes.

Il faut noter que les codes qui s'appuient sur des choix effectués au sein des mises en œuvres des fonctionnalités s'appuient en plus sur des choix effectués de manière externe. Cela est dû au fait que les choix effectués par une mise en œuvre ne peuvent concerner que des optimisations spécifiques à cette mise en œuvre et ne permettent pas de choisir parmi les mises en œuvre disponibles.

	mise en œuvre	intermédiaire	externe
FRACTAL et CCA	X		X
GRIDCCM	X		X
partage de données	X		X
fermes de tâches			X
communication collective		X	

TABLE 3.3 – Responsabilité des choix d'optimisation dans les diverses approches présentées

3.3 Analyse

Cette section présente un ensemble de critères pour l'évaluation des approches destinées à la mise en œuvre de fonctionnalités dédiées au calcul à haute performance. Les différentes catégories d'approche précédemment présentées sont évaluées au regard de ces critères.

3.3.1 Exhaustivité

Un premier critère concerne la possibilité de décrire l'intégralité des fonctionnalité requises à l'aide de l'approche choisie. Il faut noter qu'en plus des paradigmes habituels décrits en 2.1, certaines fonctionnalités concernent une classe restreinte d'applications. Le nombre de fonctionnalités qu'il peut être intéressant de supporter est donc potentiellement élevé et leur énumération exhaustive dépend de la connaissance pointue des nombreux paradigmes autour desquels diverses variations peuvent exister. De ce fait, il est intéressant pour une approche de permettre l'extension du modèle de composants *a posteriori* plutôt que de se limiter à permettre la mise en œuvre d'un ensemble fixe de fonctionnalités.

L'utilisation d'approches dans lesquelles le code de mise en œuvre est intégré soit dans les composants soit dans l'assemblage restreint les fonctionnalités qui peuvent être mises en œuvres à celles qui peuvent être projetées sur les fonctionnalités offertes par le modèle de composants. Par exemple, un modèle de composants entièrement local qui n'offrirait pas la possibilité de communiquer au travers du réseau ne permettrait pas la mise en œuvre de fonctionnalités reposant sur le réseau par cette approche. Ce problème ne devraient toutefois pas être insurmontable si le modèle de composants initial est suffisamment complet quant aux fonctionnalités de base exposées. À l'inverse, l'utilisation d'une approche dans laquelle le code est distribué avec l'exécutif ne pose pas ce type de problème mais il impose la mise à jour de l'exécutif pour chaque nouvelle fonctionnalité utilisée et implique une multiplication de l'effort de développement si plusieurs exécutifs indépendants cohabitent.

Le choix de l'étape à laquelle sont effectués les choix d'optimisation n'a pas d'influence particulière sur la possibilité de décrire la totalité des fonctionnalités requise ou sur la possibilité d'ajouter de nouvelles fonctionnalités *a posteriori*.

Par contre la responsabilité des choix est un point important puisque l'introduction de nouvelles extensions introduit par la même occasion de nouveaux paramètres. L'utilisation d'un outil externe qui existait avant l'introduction de ces nouveaux paramètres et n'en connaît donc pas la signification n'est alors pas envisageable. Les deux autres approches identifiées permettent de résoudre ce problème en fournissant des informations pour que l'outil externe puisse effectuer les choix automatiquement ou en permettant de distribuer le code qui effectue les choix en même temps que les extension.

3.3.2 Compatibilité entre extensions

Un second critère important est la possibilité d'utiliser dans une même application plusieurs fonctionnalités éventuellement développés indépendamment. Il devrait ainsi être possible d'utiliser des composants utilisant des fonctionnalités différentes au sein d'un unique assemblage, mais aussi de mettre en œuvre un unique composant reposant sur plusieurs fonctionnalités différentes.

Pour ce critère, les approches qui reposent sur l'introduction de code additionnel dans les composants en s'appuyant sur une chaîne de compilation spécifique à une fonctionnalité pose problème puisqu'il ne semble pas simplement envisageable d'utiliser deux chaînes de compilation différentes pour un unique composant. De la même manière, il ne semble pas réaliste d'utiliser au sein d'une unique application plusieurs exécutifs du modèle. Les approches qui s'appuient sur la distribution du code de mise en œuvre avec l'exécutif rendent donc difficile la cohabitation de deux fonctionnalités mises en œuvre indépendamment dans deux exécutifs différents. Finalement l'approche qui consiste à insérer le code sous la forme de nouveaux composants dans l'assemblage semble intéressante puisque les composants sont justement conçus pour pouvoir être remplacés par un équivalent compatible. Il semble envisageable de limiter les transformations possibles pour elles puissent être appliquées sur un même assemblage jusqu'à ce que l'intégralité des concepts aient été transformés. Cet aspect joue aussi en faveur de l'utilisation d'approches basées —comme celle utilisée pour la mise en œuvre des opérations de communication collective— sur des concepts additionnels de bas niveau partagés par plusieurs extensions développées au sein même du modèle de composants.

Comme cela a été noté précédemment, l'utilisation d'une approche reposant sur une modification de l'assemblage de composant empêche les choix d'être effectués lors de la compilation des composants. Le choix parmi les deux autres options (optimisation au déploiement ou au cours de l'exécution) n'ont pas d'influence particulière sur la possibilité d'utiliser des fonctionnalités développées indépendamment au sein d'une même application.

De la même manière, les différentes catégories d'approches concernant la responsabilité des choix d'optimisation n'influent pas sur la possibilité d'utiliser plusieurs extensions indépendantes tant que les choix sont indépendants. Il semble cependant probable que les paramètres d'optimisation des diverses fonctionnalités tendent à interagir. Dans ce cas, l'utilisation d'une approche dans laquelle l'attribution des valeurs des paramètres pour chaque fonctionnalité est fait dans du code qui lui est spécifique sans possibilité d'analyse des combinaisons entre les paramètres des diverses fonctionnalités peut poser problème. Il semble donc intéressant de laisser les choix être effectués par un outil externe aux mises en œuvre de fonctionnalités, éventuellement en s'appuyant sur des informations fournies par celles-ci.

3.3.3 Qualité des fonctionnalités

Un troisième critère concerne la «qualité» des mises en œuvre de fonctionnalités qu'il est possible de fournir en utilisant l'approche choisie. Tout d'abord, il s'agit de conserver les points forts des modèles de composants classiques en offrant une interface d'utilisation intuitive qui conserve le comportement «boîte noire» des composants ainsi que la séparation du code métier du code de gestion. De plus, il est nécessaire de fournir un modèle adapté au calcul à haute performance en impliquant des sur-coûts minimums et en permettant l'utilisation d'algorithmes adaptés aux ressources d'exécutions utilisées.

De ce point de vue, les approches fonctionnant en insérant le code de mise en œuvre des fonctionnalités à l'intérieur des composants a l'avantage de permettre de spécifier des API dédiées. Elles permettent aussi de limiter les sur-coûts dûs à l'utilisation d'un modèle de composants en permettant à certaines parties du code d'être appelées directement depuis les mises en œuvre de composant sans passer par le mécanisme de ports. Ces approches donnent cependant des composants dont l'interface n'est plus significative puisque les ports qu'ils exposent sont ceux nécessaire pour la mise en œuvre et non ceux logiquement exposés complexifiant ainsi leur réutilisation.

À l'inverse, les approches fonctionnant par insertion d'instances de composants dans l'assemblage limites les possibilités d'API à celles qui peuvent être exprimées sous la forme de ports dans le modèle cible. Elles impliquent aussi que les sur-coûts dûs à l'utilisation de ports doivent être payés à chaque utilisation de la fonctionnalité. Ces problèmes sont cependant nettement réduits dans les modèles qui proposent des ports correspondant à

des appels de fonctions locales impliquant des sur-coûts minimaux et des possibilités d'API exprimables étendues. Ces approches ont aussi l'avantage de conserver une interface des composants similaire à l'interface logique du composant.

Finalement, les approches reposant sur la modification de l'exécutif du modèle sont les plus intéressantes pour ce critère en permettant l'utilisation d'API dédiées, en n'impliquant aucun sur-coût et en permettant de laisser les nouveaux concepts apparaître dans l'interface des composants.

L'étape à laquelle ont lieux les choix d'optimisation ainsi que l'entité qui en est responsable influent sur la possibilité d'adapter les algorithmes aux ressources d'utilisation disponibles. Si ces choix sont effectués lors de la compilation des composants, ils ne pourront pas être adaptés aux ressources d'exécutions utilisées qui ne sont pas connues à cette étape. S'ils sont effectués lors du déploiement de l'application, ils permettent de l'adapter aux ressources d'exécutions si les ressources et l'application restent suffisamment stables au cours de l'application. Cela peut poser problème dans le cas d'applications qui évoluent au cours du temps telles que celles qui s'appuient sur des flux de travail ou dans le cas de ressources fortement volatiles comme c'est le cas des grilles et en particulier des grilles de PC par exemple. À l'inverse, les approches dans lesquelles les choix d'optimisations sont effectuées à l'exécution rendent complexe l'application conjointe de ces choix d'optimisation avec les choix de placement des composants utilisateurs sur les ressources d'exécution classiquement effectués lors du déploiement.

En ce qui concerne la responsabilité des choix, si ceux-ci sont embarqués dans le code de mise en œuvre des fonctionnalités, il n'est alors plus possible de changer la manière dont ils sont effectués en changeant par exemple les objectifs poursuivis. Dans le cas où les fonctionnalités exposent les informations sur lesquelles se basent ces choix, il est possibles que toutes les informations nécessaire à l'utilisation d'un nouvel objectif ne soient pas disponibles. De ce point de vue, les approches qui s'appuient uniquement sur un outil externe sont donc les plus adaptées.

3.3.4 Adaptation à de nouvelles ressources d'exécution

Un point important pour permettre l'utilisation d'algorithmes adaptés aux ressources d'exécutions concerne la possibilité de changer la mise en œuvre des fonctionnalités après le développement des composants. Comme cela a déjà été étudié, la durée de vie des codes de calcul scientifique est bien souvent nettement supérieure à celle des architectures d'exécution utilisée pour les exécuter. Afin de permettre l'utilisation de mise en œuvre adaptés à ces nouvelles architectures, il est donc important de permettre le changement *a posteriori* de la mise en œuvre utilisée par un composant donné.

De ce point de vue, les approches fonctionnant par distribution du code de mise en œuvre au sein des composants limitent les choix de la mise en œuvre à utiliser à un choix entre celles qui ont été intégrées dans le composant, limitant ainsi l'adaptation à d'éventuelles nouvelles architectures de ressources d'exécution. À l'inverse les deux autres approches pour lesquelles le code de mise en œuvre est distribué indépendamment des composants ne pose pas ce type de problème.

De la même manière, les approches qui reposent sur des choix effectués à la compilation des composants limitent leur adaptation à des architectures de ressources d'exécution nouvelles. Celles dans lesquelles les choix sont fait au déploiement ou au cours de l'exécution ne posent pas de tels problèmes.

De la même manière, les approches dans lesquelles les mises en œuvre sont responsables des choix d'optimisation posent problème parce que leurs algorithmes de choix risquent de ne pas être adaptés aux nouvelles ressources d'exécution. A l'inverse, l'utilisation d'algorithmes de choix externe permet d'utiliser divers variantes adaptés à divers types de ressources d'exécution. Si les mises en œuvre ne fournissent aucune information pour guider ces choix, un outil externe sera cependant limité et ne pourra pas effectuer des choix spécifiques à une mise en œuvre pour laquelle il n'a pas été initialement conçu.

3.3.5 Simplicité de mise en œuvre

Un dernier point concerne la complexité de mise en œuvre des différentes fonctionnalités dans l'approche choisie. Afin de supporter efficacement un large éventail de fonctionnalités il semble important que leurs mises en œuvre puissent être effectuées par les spécialistes de ces paradigmes. Dans ce but, il est important que la mise en œuvre d'une fonctionnalité ne nécessite qu'un minimum de connaissances spécifiques au modèle de composants dans laquelle elle est effectuée.

De ce point de vue, les approches qui s'appuient sur une modification des exécutifs du modèle posent problème puisqu'elles nécessitent une connaissance pointue de ces codes souvent complexes. L'écriture de code destiné à être inséré soit dans les composants soit sous forme de composant dans l'assemblage ne pose pas ce type de problème. La modification de la chaîne de compilation et dans une moindre mesure le développement d'un outil de transformation de l'assemblage sont cependant des tâches relativement complexes. Les approches qui s'appuient sur le développement des mises en œuvre entièrement dans le modèle de composants comme celles utilisées pour la mise en œuvre des opérations de communication collective dans CCM sont les plus intéressantes pour cette aspect.

L'étape à laquelle les choix d'optimisation sont effectués n'a pas d'influence important sur la complexité de développement d'une mise en œuvre de fonctionnalité.

Les différentes catégories pour ce qui est du code responsable de ces choix peut au contraire influencer la simplicité de développement. Les approches qui reposent sur du code d'optimisation intégré aux mises en œuvre nécessitent que celles-ci découvrent les ressources d'exécution sur lesquelles elles vont s'exécuter, ce qui peut nécessite un support dans le modèle cible. Le choix d'une approche où les mises en œuvre fournissent des informations quant à l'influence des divers paramètres rend complexe le développement du code qui effectue les choix. Un solveur de contrainte qui rechercherait la solution optimale risque de prendre un temps prohibitifs et les différentes heuristiques qui peuvent être utilisées pour accélérer ces choix sont spécifiques à des fonctionnalités particulières. Finalement l'utilisation d'un outil de choix externe sans aucun support dans les mises en œuvre offre la solution la plus simple du point de vu de la mise en œuvre.

3.3.6 Résumé

Les approches qui s'appuient sur l'insertion de code dans les composants sont intéressantes pour les faibles sur-coûts qu'elles engendrent tout en permettant d'offrir une API adaptée. Elles limitent cependant la possibilité de supporter de manières exhaustive les fonctionnalités dédiées au calcul à haute performance sous la forme d'extensions pouvant être développées de manière indépendante. Elles rendent aussi difficilement analysable l'interface des composants et ne permettent pas leur réutilisation sur des architectures de ressources différentes de celles initialement prévues et sont relativement complexe à mettre en œuvre.

Les approches basées sur une distribution du code avec l'exécutif du modèle de composants sont adaptées au développement de fonctionnalités sous la forme d'extensions mais ne permet pas à ces extensions d'être développées de manière indépendantes. Elles permettent aussi d'offrir une API intuitive aux utilisateurs et de s'adapter à d'éventuelles nouvelles ressources d'exécution. Cette possibilité implique cependant une complexité de mise en œuvre élevée et de la nécessité de distribuer un nouvel exécutif pour chaque nouvelle fonctionnalité ou catégories de ressources d'exécution supportée.

Finalement les approches basées sur la distribution du code sous la forme de composants insérés dans l'assemblage permettent le développement indépendant d'extensions offrant de nouvelles fonctionnalités ou de nouvelles mises en œuvre de fonctionnalités existantes permettant de gérer d'éventuelles nouvelles architectures des ressources d'exécution. Elles sont relativement simple à mettre en œuvre mais nécessitent pour ne pas impliquer de sur-coûts élevés et pour permettre d'offrir une interface d'utilisation intuitive d'être basées sur un modèle de composants supportant des interactions de bas niveau entre composants.

Les approches dans lesquelles les choix d'optimisation sont effectuées à la compilation des composants ne permettent pas de s'adapter pleinement aux ressources d'exécution disponibles puisque celles-ci ne sont pas connues à cette étape. Il faut noter qu'il n'existe d'ailleurs pas d'approche qui s'appuie uniquement sur des choix effectués à cette étape.

Les approches dans lesquelles ces choix sont fait lors du déploiement permettent de s'adapter aux ressources d'exécution disponibles mais limitent une éventuelle adaptation dynamique si l'application ou les ressources disponibles évoluent.

Finalement, les approches dans lesquelles ces choix sont effectués au cours de l'exécution de l'application permettent de telles adaptation dynamiques mais restreignent la possibilité d'effectuer des choix concertés avec ceux concernant le déploiement des composants utilisateurs sur les ressources d'exécution.

Les approches dans lesquelles le code de mise en œuvre des fonctionnalités est aussi responsable des divers choix d'optimisation permettent le développement de nouvelles fonctionnalités de manière indépendante mais limitent le développement de nouvelles mise en œuvre de fonctionnalités existantes. Elles imposent aussi des critères de choix figés qui peuvent ne pas correspondre à ceux poursuivis par l'utilisateur ou aux ressources d'exécution disponibles. Finalement elles limitent la possibilité d'effectuer des choix concertés entre les paramètres de plusieurs fonctionnalités utilisées au sein d'une même application.

Les approches dans lesquelles les choix sont effectués par un outil externe guidé par des informations fournies par les mise en œuvres sont intéressantes car elles supportent à la fois le développement de nouvelles fonctionnalités et de nouvelles mise en œuvre de fonctionnalités existantes de manière indépendante. Elles permettent aussi de s'adapter à l'apparition de nouveaux type de ressources d'exécution ou de nouveaux objectifs poursuivis. Dans ce but, le choix des informations exposées par les mises en œuvre est toutefois un problème complexe. Des informations trop spécifiques risquent de plus être utilisables pour poursuivre des objectifs différents ou dans le cadre de ressources d'exécutions différentes de celles pour lesquelles elles ont été pensées. À l'inverse, des informations trop générales risquent de rendre difficile leur utilisation en nécessitant des outils excessivement complexe pour les traiter.

Finalement les approches qui s'appuient uniquement sur un outil externe offrent une solution relativement simple à mettre en œuvre permettant d'utiliser des heuristiques existantes adaptées à divers objectifs ainsi qu'à diverses architectures de ressources. Ces approches deviennent cependant inutilisables dans le cas où des fonctionnalités et des mises en œuvre de ces fonctionnalités pour lesquelles elles n'ont pas été conçues sont utilisées. Elles s'avèrent donc inadaptées à l'extension de modèles de composants existant avec de nouvelles fonctionnalités.

3.4 Présentation de notre approche

Les sections précédentes ont présenté l'état de l'art pour ce qui est des approches utilisées dans la mise en œuvre de fonctionnalités dédiées au calcul à haute performance dans les modèles de composants classiques, proposé une classification de ces approches selon trois critères et analysé les avantages et inconvénients de chaque catégorie d'approche. Nous présentons dans cette section l'approche que nous proposons pour permettre le support des applications de calcul scientifique dans les modèles de composants. Il s'agit de proposer un modèle de composants qui simplifie la mise en œuvre de fonctionnalités dédiées à ces applications.

Pour cela, cette section commence par présenter l'approche que nous nous proposons d'adopter pour le développement de ces fonctionnalités. Elle s'intéresse ensuite aux propriétés que se doit de posséder un modèle de composants pour supporter cette approche. Finalement, elle identifie les problèmes qui doivent être résolus pour proposer un tel modèle.

3.4.1 Une approche basée sur le concept de bibliothèque

Afin de développer des fonctionnalités dédiées a calcul à haute performance dans les modèles de composants, nous avons vu que plusieurs approches pouvaient être utilisées. L'analyse que nous avons présenté montre qu'afin de supporter un grand spectre de fonctionnalités sur un grand spectre de ressources d'exécution il est important de permettre le développement de ces fonctionnalités sous la forme d'extensions. Nous avons aussi étudié comment ces fonctionnalités peuvent être mises en œuvre dans les modèles de composants par l'ajout de nouveaux concepts dans ces modèles.

Pour faire un parallèle avec les langage de programmation classiques, ces approches reviennent à étendre le langage et sa mise en œuvre (le compilateur) pour supporter de nouveaux concepts. Dans ce cadre, c'est par exemple l'approche utilisée pour supporter le parallélisme dans OpenMP.

Une approche parmi celles présentées constitue cependant un cas particulier. Il s'agit de l'approche utilisée pour mettre en œuvre le support des opérations de communication collective. Dans ce cas le modèle de composants est aussi étendu, mais —contrairement aux autres approches— il n'est pas directement étendu avec le concept mis en œuvre. À la place, des concepts de plus bas niveau sont introduits, les composants répliquants et les connexions `AnyToAny`. La mise en œuvre des opérations de communication collective est alors effectuée sans modification supplémentaire du modèle en s'appuyant sur ces concepts.

Pour faire un nouveau parallèle avec les langages de programmation classiques, cette approche revient à utiliser le langage pour mettre en œuvre des fonctionnalités additionnelles, c'est le concept de bibliothèque. Comme nous l'avons vu dans le chapitre 2, cette approche est utilisée pour la mise en œuvre de nombreuses fonctionnalités dédiées au parallélisme dans les langages de programmation classiques.

Une telle approche simplifie le développement de fonctionnalités sous la forme d'extension puisqu'il s'agit alors simplement d'offrir une nouvelle bibliothèque pour chaque nouvelle fonctionnalité. Elle permet aussi de s'adapter à d'éventuelles nouvelles catégories de ressources d'exécution par le développement de nouvelles mises en œuvre des bibliothèques utilisées. Elle assure la compatibilité entre les extensions qui s'appuient toutes sur le même ensemble de fonctionnalités de base fournies par le modèle. Enfin, en ne nécessitant pas de modification profonde du modèle, cette approche simplifie grandement la mise en œuvre des fonctionnalités.

Il faut toutefois noter que pour utiliser cette approche pour la mise en œuvre des opérations de communication collective, le modèle a malgré tout été modifié. Cela est dû au fait que les modèles de composants existants ne proposent pas de concept de bibliothèque en tant que tel. Le concept correspondant dans ces modèles est celui de composant. Ces derniers n'offrent cependant pas de support satisfaisant pour la mise en œuvre de fonctionnalités dédiées au calcul à haute performance.

Contrairement au cas des langages de programmation, la localité de l'exécution d'un code est déterminée dans les modèles de composants par le composant qui fournit le code et non par l'appelant. Cet aspect limite la possibilité pour un composant de gérer la distribution de l'exécution d'un programme qui se trouve au contraire encodée dans l'assemblage de composant. Les langages de description d'assemblage sont aussi très statiques et ne permettent donc pas la description d'assemblages qui s'adapteraient aux ressources d'exécution.

De plus les composants ne peuvent gérer la distribution ou la réplication des mises en œuvres de services que pour ceux fournis par des instances de composants qu'ils contiennent dans le cas des composites et non par ceux qu'ils utilisent au travers de ports. Cet aspect signifie qu'un composite qui définirait un assemblage d'instances de composants adapté à un type de ressources d'exécution donné devra être ré-écrit pour chaque nouveau service auquel il s'applique, anéantissant de fait sa possible utilisation comme bibliothèque.

Finalement, dans la majorité des modèles, chaque composant possède une unique mise en œuvre. Des modèles tels que CCM permettent aux composants primitifs de posséder une mise œuvre dédiée à chaque architecture de processeur sur laquelle ils peuvent s'exécuter. Cela ne permet toutefois pas de choisir la mise en œuvre d'un composant la plus adaptée

en fonction d'autres critères limitant ainsi la possibilité d'adaptation des applications aux ressources d'exécution disponibles.

Ces limitation nous amènent à étudier dans la suite de cette section les besoins pour permettre la mise en œuvre de fonctionnalités dans les modèles de composants sous la forme de bibliothèques.

3.4.2 Fonctionnalités pour le support de bibliothèques dans les modèles de composants

Le chapitre 2 a montré que les fonctionnalités dédiées au calcul à haute performance introduisent des concepts de deux types :

– de nouvelles manières de mettre en œuvre les composants ; et
– de nouvelles formes d'interactions entre composants.

La description de nouvelles mises en œuvre de composants dans des modèles extérieurs est possible dans la majorité des modèles de composant. L'aspect qui n'est pas géré concerne les cas où une partie du comportement du composant est fixée et une partie seulement doit être spécifiée par l'utilisateur. Il s'agit en fait de permettre la définition au sein du modèle de nouveaux squelettes algorithmiques et de leurs mises en œuvres.

La description de nouvelles formes d'interactions nécessite quant à elle d'identifier les différents types d'interactions entres composants à l'aide d'un concept dédié. C'est ce que propose le concept de connecteur au sein des langage de description d'architecture (*Architecture Description Languages* – ADL). Certains travaux proposent cependant de l'intégrer aux modèles de composants et de permettre la définition de nouveaux connecteurs et de leur mise en œuvre. C'est par exemple le cas du modèle de composants SOFA [11] ainsi que d'un modèle proposé par Matougui et Beugnard [43].

Ces deux aspects permettent de décrire les applications à un plus haut niveau d'abstraction en décrivant les opérations de composition utilisées et en laissant leur mise en œuvre être proposée indépendamment. Pour pouvoir offrir des fonctionnalités qui permettent l'adaptation des applications aux ressources d'exécution, il est cependant nécessaire de proposer plusieurs mises en œuvre de ces formes de composition. En plus de la possibilité de décrire des squelettes algorithmiques et des connecteurs dans le modèle il est donc nécessaire de pouvoir en proposer plusieurs mises en œuvres pour en choisir celle qui est la plus adaptée aux ressources d'exécution effectivement utilisées.

Développement de squelettes Comme présentés dans la section 2.2.3, les squelettes algorithmiques décrivent la structure de classes d'algorithmes récurrents du parallélisme. A la manière des fonctions d'ordre supérieure [24], ils ne fournissent pas le code de mise en œuvre final qui est fourni en tant que paramètre lors de l'instancitation du squelette.

Le développement de squelettes directement dans un modèle de composants nécessite donc la possibilité de décrire des assemblages d'ordre supérieur. Il s'agit de spécifier la forme d'un assemblage de composants —c'est à dire les instances et leur inter-connections— mais à laisser leur type être spécifié *a posteriori*.

On peut s'intéresser à l'exemple du squelette de ferme de tâches présenté sur la figure 3.3. Une mise en œuvre de cette ferme de tâches en tant que composant dans un modèle classique sera typiquement constituée d'un composite contenant des instances de trois types distincts de composants utilisés comme distributeur, travailleurs et collecteur. Le résultat est un composant fortement lié aux types des données traitées, au calcul qui leur est appliqué et au nombre de fils d'exécution pouvant être exécutés en parallèle par les ressources d'exécution.

Il semble alors intéressant de permettre aux types des composants contenus dans la ferme de tâches ainsi que ceux des données traitées d'être passées en paramètre d'un tel composite. C'est ce que propose le concept de généricité [45]. Le nombre d'instances de travailleurs à utiliser est une valeur entière qui pourrait être passée comme paramètre dans les modèles de composants existant en s'appuyant sur le concept d'attribut de composant.

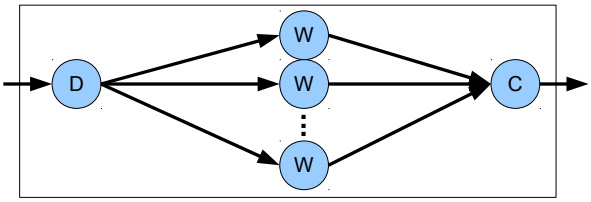

FIGURE 3.3 – Le squelette de ferme de tâches comportant un distributeur (D) qui répartie le flux de données en entrée entre les travailleurs (W) qui effectuent le calcul et un collecteur (C) qui ré-assemble le flux en sortie.

La mise en œuvre de squelettes algorithmiques peut donc s'appuyer sur les concepts de hiérarchie et de généricité comme nous le verrons dans le chapitre 4. La hiérarchie permet de définir des mises en œuvres composites pour les composants. La généricité permet de leur passer en paramètre le type de certaines instances internes ou des éléments qui influent sur leur structure.

Cependant, ces éléments ne sont pas suffisant. En effet nous avons noté qu'il est nécessaire de pouvoir adapter la mise en œuvre des squelettes aux ressources d'exécution disponibles. Puisque avec cet approche les squelettes sont des composants particuliers, il est nécessaire qu'ils puissent avoir plusieurs mises en œuvres.

Développement d'interactions Le concept de connecteur est relativement similaire à celui de composant. Il s'agit d'une entité qui comporte un ensemble de rôles que peuvent remplir les composants qui participent à une interaction. Comme les composants, les connecteurs peuvent être mis en œuvre dans un modèle externe ou bien par un assemblage.

Les connecteurs introduisent donc la possibilité de décrire des interactions qui font intervenir plus de deux participants. Dans le cadre de modèle hiérarchiques, il est possible que les instances de composant primitives qui participent à ces interactions soient réparties à plusieurs niveaux de la hiérarchie. Cette possibilité implique des questions quant à la manière dont est décrite l'interface des composants.

Comme nous le verrons dans le chapitre 5, l'approche habituelle qui consiste à décrire cette interface à l'aide de ports est problématique. Elle impose de choisir entre la possibilité de choix de mise en œuvre des composants et la possibilité de décrire des mises en œuvre efficaces des interactions. Il est donc nécessaire de proposer une nouvelle approche pour la description de l'interface des composants dans ce cadre.

3.4.3 Mise en œuvre de la généricité et des connecteurs dans un modèle de composants

La mise en œuvre de ces concepts nécessite d'être supporté par le modèle de composants. Afin de limiter le sur-coût dû à leur utilisation et pour les raisons présentées dans la section 3.3, nous proposons de nous appuyer sur une transformation de l'assemblage. Une telle transformation permet d'optenir comme résultat un assemblage de composants ne comportant plus de concept avancés mais uniquement des composants primitifs.

Pour appliquer cette transformation, les outils de développement basés sur les modèles [56] et notamment ceux qui supportent la transformation de modèles offrent une approche intéressante. L'ingénierie dirigée par les modèles est une approche qui augmente le niveau d'abstraction en se focalisant sur la modélisation de concepts spécifiques au domaine étudié plutôt qu'à l'aspect calculatoire.

Dans ces approches, les objets du monde réel sont modélisés par des applications qui sont elle même modélisées à l'aide de modèles spécifiques au domaine qui sont décrits dans un modèle du modèle ou métamodèle. En décrivant les divers modèles spécifiques au

domaine à l'aide d'un même métamodèle il devient possible d'automatiser la transformation d'instances d'un modèle donné en instances d'un autre modèle.

Par exemple, l'architecture dirigée par les modèles [47] (*Model Driven Architecture* – MDA) telle que proposée par l'OMG s'appuie sur de telles transformations. Cette approche consiste à transformer les descriptions d'applications effectuées dans des modèles spécifiques au domaine vers des modèles de moins en moins spécifique au domaine mais de plus en plus spécifique à la plateforme d'exécution en s'appuyant sur une description de cette plateforme.

Cette approche est intéressante dans notre cas puisqu'elle facilite la transformation d'une description des applications comportant des concepts abstrait tels que des composants génériques ou des connecteurs en application où ces concepts ont été remplacés par leurs mises en œuvre adaptées aux ressources d'exécution disponibles.

Ces transformations ainsi que les choix d'optimisation associés peuvent être effectués soit lors du déploiement soit au cours de l'exécution. De plus, le code responsable de ces choix d'optimisation peut être distribué avec les extensions ou bien de manière séparée et s'appuyer sur des informations des informations exposées par les extensions.

Sur ces deux aspects, les arguments en faveur de l'une ou l'autre des approches semblent tous pertinents. Nous nous proposons donc de fournir une mise en œuvre offrant la possibilité d'explorer chacune de ces possibilités de manière expérimentale.

3.5 Conclusion

Dans ce chapitre, nous avons présenté les diverses approches qui ont été utilisées pour étendre les modèles de composants avec des fonctionnalités dédiées au parallélisme. Nous avons ensuite effectué une classification de ces approches selon trois critères, la localité du code qui met en œuvre ces fonctionnalités, l'étape à laquelle les diverses optimisations possibles sont effectuées et la responsabilité de ces choix. Nous avons ensuite analysé les différentes catégories d'approche au regard des objectifs poursuivi. Finalement nous avons présenté notre approche pour permettre la mise en œuvre de fonctionnalités sous la forme de bibliothèques.

L'approche proposée s'appuie sur la présence au sein du modèle de composants des concepts de hiérarchie, de généricité et de connecteurs ainsi que sur la possibilité de choisir la mise en œuvre des composants et des connecteurs. Le concept de généricité appliqué aux modèles de composants logiciels n'est cependant pas clairement défini, son introduction constitue notre première contribution présentée dans le chapitre 4.

L'introduction du concept de connecteur dans les modèles de composants logiciels a déjà été proposée, mais ses interactions avec les concepts de hiérarchie et de choix de mises en œuvre n'ont pas été étudiées. L'étude des problèmes liés à ces interactions et la proposition d'une nouvelle approche pour la description de l'interface des composants qui les résout constitue notre seconde contribution présentée dans le chapitre 5.

Le modèle finalement proposé comporte les concepts de hiérarchie, de généricité, de connecteur comme entité de première classe et de choix de mises en œuvres. Il s'agit de l'ensemble des concepts dont nous postulons qu'ils permettent de définir de nouveaux opérateurs de composition (squelettes algorithmiques et interactions) sous la forme de bibliothèque.

Une approche pour la mise en œuvre de tous ces concepts consiste à s'appuyer sur une transformation de l'assemblage lors du déploiement. Une méthode qui offre des concepts intéressant pour effectuer ce type de transformation est issue de l'ingénierie dirigée par les modèles. C'est cette méthode qui est utilisée dans la plate-forme de mise en œuvre qui constitue notre troisième contribution et qui est présentée dans le chapitre 6.

Chapitre 4

Description de squelettes algorithmiques dans un modèle de composants

Le chapitre précédent a présenté l'approche que nous proposons pour concevoir un modèle de composants extensible par des fonctionnalités dédiées au calcul à haute performance. Cette approche est caractérisée par la description des applications dans un modèle qui offre un haut niveau d'abstraction et une transformation vers un modèle proche du matériel lorsque les ressources d'exécution sont connues. Un des besoins identifiés pour le modèle de haut niveau concerne l'utilisation de squelettes algorithmiques pour permettre la description de mises en œuvre de composants qui s'adaptent aux ressources d'exécution disponibles.

La contribution présentée dans ce chapitre consiste en l'étude et la mise en œuvre d'une approche pour supporter la description de squelettes algorithmiques et leur utilisation comme mise en œuvre dans un modèle de composants. L'approche proposée s'appuie sur l'ajout du concept de généricité similaire aux notions de *generics* en Java ou de *templates* en C++.

La section 4.1 propose une première analyse du problème, les travaux existant y sont présentés, les concepts liés à la généricité isolés et les approches qui peuvent être adoptées pour mettre en œuvre la généricité discutées. La section 4.2 présente ensuite notre approche pour la mise en œuvre du concept de généricité qui comporte un modèle abstrait et un algorithme qui permet de transformer une application décrite dans ce modèle qui s'appuie sur la généricité en une application sémantiquement équivalente où le concept de généricité a été ôté. La section 4.3 applique cette approche au modèle de composants SCA pour valider l'approche. Finalement, la section 4.4 conclue le chapitre.

4.1 Analyse préliminaire

Le chapitre 3 a présenté notre hypothèse selon laquelle le support des concepts de hiérarchie et de généricité dans les modèles de composants logiciels permettraient la définition de composants se comportant comme des squelettes algorithmiques. La notion de généricité n'a cependant pas été clairement définie pour les modèles de composants logiciels. Cette section s'attache donc à déterminer la signification de ce concept dans ce cadre.

Dans un premier temps, les modèles qui comportent la notion de généricité sont présentés ainsi que ceux qui introduisent des concepts proches dans les modèles de composants. Les modèles génériques existant sont ensuite analysés plus finement pour en extraire les concepts spécifiques de la généricité et étudier leur applicabilité aux modèles de composants logiciels. Finalement, les différentes approches pour l'introduction de ces concepts sont présentées et discutées dans le cadre des modèles de composants.

4.1.1 Analyse de l'existant

Contrairement aux langage à objet parmi lesquels beaucoup la supportent, nous ne connaissons pas de modèle de composants qui supporte entièrement la généricité. La plupart des modèles de composants supportent un concept de *propriété* qui peut être comparé à une forme minimaliste de généricité. Le seul modèle qui propose des concept plus proche de la généricité est HOC (*Higher Order Components*).

La généricité dans les langages orienté objet La généricité [45] est une fonctionnalité incontournable des langages orienté objet, comme entre autre Ada, C++, C♯, Eiffel et Java [29]. Il s'agit pour les classes, les méthodes et parfois les procédures d'accepter des types comme paramètres. Une utilisation classique de la généricité consiste à mettre en œuvre des conteneurs pour lesquels le type des données contenues est un paramètre.

Les éléments qui peuvent être utilisés comme valeur de paramètres varient en fonction des langages. C++ supporte l'utilisation comme paramètre de tous les types de données ainsi que de valeurs énumérées (entiers et énumérations principalement). Java ne permet d'utiliser comme paramètre que des classes à l'exclusion des types primitifs ou des valeurs de données.

Les valeurs utilisées comme paramètres de méthodes ont généralement un domaine de validité spécifié, par exemple à l'aide d'un type associé au paramètre. De manière similaire, certains langages proposent des constructions pour restreindre le domaine de validité des types pris en paramètre de constructions génériques. Par exemple il est possible de définir en Java des contraintes d'héritage pour les types pris en paramètre d'une construction générique [17]. Dans d'autres langages comme C++, les contraintes sont implicites [60] et c'est les utilisations de ces paramètres qui déterminent leur valeur possible.

On peut aussi noter la présence dans certains langages comme C++ du support de spécialisations explicites. C'est à dire qu'il est possible de proposer une mise en œuvre différente de la construction générique quand ses paramètres prennent un certain ensemble de valeurs. Il est alors possible de fournir des mises en œuvre optimisées pour certain types.

De plus, cette fonctionnalité rend le langage Turing-complet et permet la méta-programmation à base de *templates* [2] (*template meta-programming*). Il s'agit de s'appuyer sur le fait que la les éléments génériques sont traités à la compilation pour faire exécuter des programmes à cette étape.

Les *propriétés* de composants Le concept de *propriétés* de composant constituent la fonctionnalité la plus proche de la généricité trouvée dans les modèles de composants classiques. Il s'agit de paramètres qui peuvent être utilisés pour configurer le comportement des composants. Dans certains modèles tels que CCM, ces paramètres peuvent voir leur valeur évoluer au cours de l'exécution de l'application. Dans d'autres modèles comme par exemple

CCA, leur valeur ne peut plus changer une fois définie dans l'assemblage de composants les rendant en cela similaires aux paramètres de constructions génériques.

Il faut cependant noter qu'à la différence de véritables paramètres de constructions génériques, ces propriétés sont limitées à la transmission de valeurs de données et non pas de types de données, de ports ou de composants.

Higher Order Components HOC [31] est un un projet basé sur l'intergiciel (*middleware*) de grille GLOBUS et qui introduit un certain degré de généricité. HOC propose un mécanisme de *factory* qui permet à une mise en œuvre de service GLOBUS de créer des instances d'autres services dont la mise en œuvre est spécifiée par un identifiant. Pour les développeurs de HOC, ce mécanisme permet de décrire des squelettes algorithmiques comme par exemple le *map*, le *pipeline*, le *scan*, le «*divide and conquer*» et la ferme de tâches qu'ils décrivent en détail. Il s'agit de définir une mise en œuvre de service qui prend en paramètre les identifiants des mise en œuvres de services à utiliser au sein du squelette.

Une première limitation de cette approche est due à l'absence de langage d'assemblage dédié. L'utilisation d'une API au sein d'un programme JAVA qui mélange l'instanciation et l'utilisation des services aux calculs utilisant leurs résultats rend compliquée l'identification des services utilisés. Notamment, puisque les identifiant de mise en œuvre de services sont de simples entiers, il est très compliqué de les différencier de paramètres entiers classiques et de vérifier la validité des mises en œuvre de services choisies à la compilation comme par exemple le fait qu'elles fournissent l'interface attendue par le squelette.

De plus, la généricité est limitée aux mise en œuvres de services, il n'est pas possible de l'appliquer aux paramètres des méthodes de l'interface d'un service par exemple. Pour la mise en œuvre de la ferme de tâches proposée par les auteurs, le résultat est l'utilisation de tableaux de `double` JAVA comme seul élément pouvant apparaître en entrée ou en sortie de la ferme de tâches. L'utilisation d'autre types de donnée nécessiterait autant de mise en œuvre que de type de donnée, ce qui restreint l'intérêt de l'approche.

4.1.2 Présentation des concepts

La suite de cette section présente les deux principaux éléments de la généricité que sont les éléments génériques et les spécialisations. Le concept de spécialisation explicite présent dans certains modèles génériques est aussi présenté. Lorsque cela est nécessaire, l'application de ces concepts aux modèles de composants logiciels est discutée.

Élément générique Un type générique est un type du modèle qui accepte d'autres types comme paramètres.

```
template <class T>
class GenClass {
private:
  T m_value;
  //...
};
```

FIGURE 4.1 – Exemple en C++ d'une classe `GenClass` générique acceptant une classe `T` comme paramètre

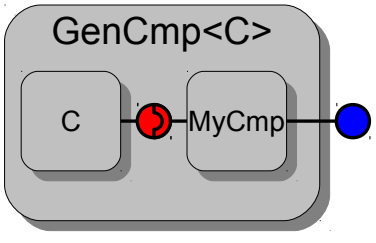

FIGURE 4.2 – Exemple d'un type de composant `GenCmp` générique acceptant un type de composant `C` comme paramètre

Dans les langages objets, une classe générique peut accepter d'autres types en paramètres, notamment des classes. L'exemple de la figure 4.1 présente la classe générique `GenClass` qui accepte une classe `T` en paramètre. Ce paramètre peut être utilisé comme

n'importe quelle autre classe dans la mise en œuvre de la classe générique. Dans l'exemple, le paramètre `T` est utilisé pour typer la variable `m_value`.

Pour adopter une approche similaire dans le monde des composants, on définit un type de composant générique comme étant un type de composant qui accepte d'autres types en paramètres, notamment des types de composants, des types de ports ou des interfaces objet. L'exemple de la figure 4.2 présente le type de composant générique `GenCmp` qui accepte un type de composant `C` en paramètre. Comme pour les objets, les types pris en paramètre peuvent être utilisés de manière similaire à leurs équivalents globaux dans la mise en œuvre du composant. Dans l'exemple, le type de composant `C` est utilisé pour typer une instance de composant du composite `GenCmp`.

Certains langages supportent l'ajout de contraintes sur les paramètres de types génériques. Par exemple en Java, il est possible de spécifier qu'une classe prise en paramètre doit être une super-classe ou une sous-classe d'une autre. Seules les utilisations du paramètre qui sont assurées d'être valides par ces contraintes sont autorisées dans la mise en œuvre de la classe générique. À l'inverse, en C++ aucune contrainte n'est explicitement spécifiée et l'utilisation d'un type comme paramètre est valide si toutes les utilisations qui en sont faites dans la mise en œuvre sont valides.

Le support de contraintes sur les types pris en paramètres semblent intéressant pour permettre au développeur d'expliciter l'utilisation prévue des types génériques. Le choix d'un ensemble de contraintes permettant d'assurer la validité de l'utilisation des paramètres comme en Java nécessiterait une étude plus approfondie. Une solution intermédiaire semble intéressante : elle consiste à permettre l'expression de contraintes mais à ne pas les utiliser pour restreindre l'utilisation des paramètres dans la mise en œuvre et à vérifier la validité de l'utilisation d'un type donné comme paramètre en fonction de ses utilisations, comme en C++.

Une autre possibilité, présente dans certains langages tel que C++, consiste à spécifier des valeurs par défaut pour les paramètres. Cette possibilité semble intéressante en permettant d'offrir un degré élevé de flexibilité tout en offrant une utilisation plus simple aux utilisateurs qui ne sont pas intéressés par toute l'étendue de la paramétrisation. Cette possibilité ne présente pas de difficulté à appliquer aux modèles de composants.

Spécialisation Contrairement à leurs contreparties classiques, les types génériques ne peuvent pas être utilisés tel-quels. Il est nécessaire d'associer une valeur à chaque paramètre. Un nouveau concept est introduit qui référence un type générique et associe une valeur à chaque paramètre. On parle de spécialisation.

```
MyClass {
private:
  GenClass<std::string> m_genval;
  //...
};
```

Figure 4.3 – Exemple en C++ de spécialisation de la classe générique `GenClass` avec `std::string` comme argument

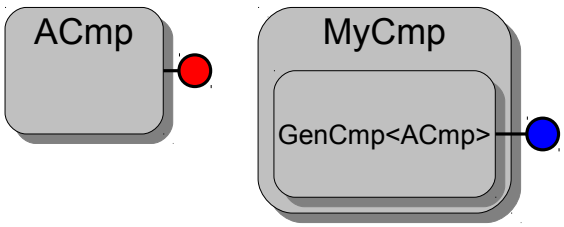

Figure 4.4 – Exemple de spécialisation du type de composant `GenCmp` avec le type de composant `ACmp` comme argument

Dans les langages objets, les spécialisations de classes génériques prennent des classes en arguments et peuvent être utilisées de manière interchangeables avec leur contreparties classiques, notamment pour typer des variables. La figure 4.3 présente un exemple où la variable `m_genval` est typée par une spécialisation de la classe générique `GenClass` de la figure 4.1 qui associe au paramètre `T` la classe `std::string`.

Pour adopter une approche similaire dans le monde des composants, on définit une spécialisation de type de composant générique comme étant un élément qui accepte d'autres

types en argument et qui peut être utilisé quand un type de composant est attendu. La figure 4.4 présente un exemple où le type de composant `MyCmp` contient dans sa mise en œuvre composite une instance de composant typée par une spécialisation du type de composant générique `GenCmp` de la figure 4.2 qui associe au paramètre `C` le type de composant `ACmp`.

Spécialisation explicite Comme nous l'avons vu, certain langages tels que C++ supportent le concept de spécialisation explicite qui permet d'offrir une mise en œuvre différente lorsque les paramètres prennent certaines valeurs. En introduisant une construction conditionnelle en plus de la récursion, cette fonctionnalité a généralement comme conséquence de rendre le système de type Turing-complet.

```
template <class P>
class GenClass<P*> {
private:
  P* m_value;
  //...
};
```

FIGURE 4.5 – Exemple en C++ de spécialisation explicite de la classe `GenClass` quand le paramètre `T` est de type «pointeur vers `P`»

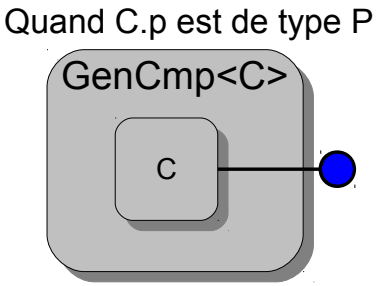

FIGURE 4.6 – Exemple de spécialisation explicite du type de composant `GenClass` quand port du type de composant paramètre `C` est de type `P`

Dans les langages objets tels que C++, il est possible de spécifier une mise en œuvre d'une classe générique qui doit être utilisée lorsque les arguments de la spécialisation satisfont certaines contraintes. La figure 4.5 présente un exemple où la classe générique `GenClass` de la figure 4.1 possède une mise en œuvre spécifique lorsque le paramètre `T` prend comme valeur un type pointeur.

Pour adopter une approche similaire dans le monde des composants, on définit le concept de spécialisation explicite de type de composant générique comme étant une mise en œuvre particulière à utiliser quand certaines contraintes sont vérifiées par les valeurs des paramètres. La figure 4.6 présente un exemple où le type de composant générique `GenClass` de la figure 4.2 possède une mise en œuvre adaptée dans le cas où le port `p` du type de composant `C` passé en paramètre est de type `P`.

4.1.3 Approche de mise en œuvre

Deux approches principales existent pour supporter la généricité. La première approche —dite par effacement de type (*type erasure*)— est par exemple utilisée pour JAVA. Elle consiste à ne compiler qu'une seule version du type générique où les types pris en paramètres sont remplacés par la valeur la plus générale possible déduite des contraintes. La seconde approche —qu'on qualifiera d'approche par compilation spécialisée— est par exemple utilisée pour C++. Elle consiste à compiler une version du type générique pour chaque spécialisation utilisée dans laquelle les types pris en paramètre sont remplacés par leurs valeurs effectives.

La première approche a l'avantage de limiter la taille du code généré par rapport à l'approche par compilation spécialisée. En effet, une seule version du code est compilée pour la première approche alors que dans la seconde, chaque spécialisation utilisée correspond à une version compilée. Elle nécessite toutefois de pouvoir manipuler les instances des divers types selon une unique interface ce qui suppose l'utilisation d'indirections alors que la seconde approche n'implique aucun sur-coût à l'exécution.

La première approche restreint toutefois les opérations qui peuvent être effectuées sur les types pris en paramètres à celles qui peuvent être effectuées en ne connaissant que la version générale pour laquelle le type est compilé. Par exemple, en Java il n'est pas possible de créer une instance d'une classe prise en paramètre puisque son type exact n'est pas connu au sein de la mise en œuvre. Cela signifie aussi qu'il est important de pouvoir déduire un type général sur lequel les opérations utilisées dans la mise en œuvre peuvent être utilisé et donc qu'il est nécessaire de s'appuyer sur des contraintes fortes pour déduire ce type.

À l'inverse, la seconde approche restreint l'utilisation des types génériques aux spécialisations qui ont effectivement été compilées. Il n'est donc notamment pas possible d'instancier dynamiquement des spécialisations de types génériques qui n'auraient pas été utilisés dans la version statique du programme.

Finalement, dans la seconde approche où chaque spécialisation est compilée séparément il est possible de choisir une mise en œuvre différente pour certaines spécialisations et donc de supporter les spécialisations explicites.

Nous avons vu précédemment que la définition de contraintes suffisamment expressives pour restreindre l'utilisation des paramètres de type génériques dans les modèles de composants n'a pas encore été proposée. Il semble aussi important de pouvoir manipuler entièrement les types de composants et notamment d'en créer des instances au sein de composites. Dans le cadre du calcul à haute performance, l'absence de sur-coûts à l'exécution est aussi un point important. Finalement la possibilité d'utiliser des spécialisations explicites et la méta-programmation qui va avec semblent importantes.

Nous étudions donc dans la suite l'application d'une approche par compilation d'assemblage. Il s'agit de proposer un processus qui étant donné un assemblage de composant comportant de la généricité génère un assemblage équivalent dans lequel il n'y a plus de généricité. Il s'agit donc de remplacer chaque utilisation d'une spécialisation d'un type générique par la compilation d'un type non générique correspondant à cette spécialisation.

4.2 Modèle

Cette section présente un modèle abstrait pour la mise en œuvre de la généricité au sein des modèles de composants. Un patron de conception pour la modélisation des concepts présentés en 4.1.2 est d'abord proposé. Un algorithme est ensuite décrit pour transformer une application qui s'appuie sur la généricité en une application sémantiquement équivalente ne comportant plus d'éléments génériques.

4.2.1 Patron de conception pour la modélisation de la généricité

Étant donné l'ensemble des concepts liés à la généricité, il est nécessaire de choisir une approche afin de les modéliser. Cette section propose un patron de conception pour introduire la généricité au sein d'un métamodèle de composants.

Pour appliquer ce patron, il faut au préalable identifier les éléments du modèle à rendre génériques et ceux à rendre utilisables comme paramètres de ces éléments. Dans cette présentation, nous nous appuyons sur un exemple où les types de composants sont rendus génériques et où les types de ports peuvent être utilisés comme paramètres.

Cœur du patron : types génériques et paramètres Le schéma UML de la figure 4.7 présente les principales méta-classes du patron de conception avec leurs inter-relations. La méta-classes `GenericTypeDefinition` représente une définition de type générique qui possède des paramètres modélisés par la méta-classe `GenericParameter`. La méta-classe `Specialization` modélise une spécialisation d'un type générique qu'elle référence et qui possède des arguments modélisés par la méta-classe `GenericArgument`. Chaque instance de `GenericArgument` référence le paramètre auquel le paramètre attribue une valeur.

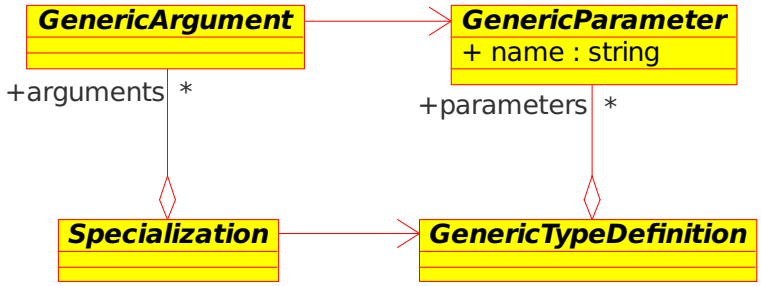

FIGURE 4.7 – Schéma UML du patron de conception pour la modélisation de la généricité

Les méta-classes `GenericParameter` et `GenericArgument` doivent être spécialisées pour chaque élément du modèle pouvant effectivement être utilisé comme paramètre de générique. Le schéma UML de la figure 4.8 présente un exemple où on permet de passer des types de port en paramètre. Dans ce cas, deux méta-classes sont créées : `PorttypeParameter` et `PorttypeArgument` qui héritent de `GenericParameter` et `GenericArgument` respectivement. De plus, une méta-classe `PorttypeParameterReference` qui hérite `PorttypeSpecification` permet d'utiliser un paramètre quand une spécification de type est attendue. La méta-classe `PorttypeArgument` référence un `PorttypeSpecification` qui représente la valeur que l'argument associe au paramètre.

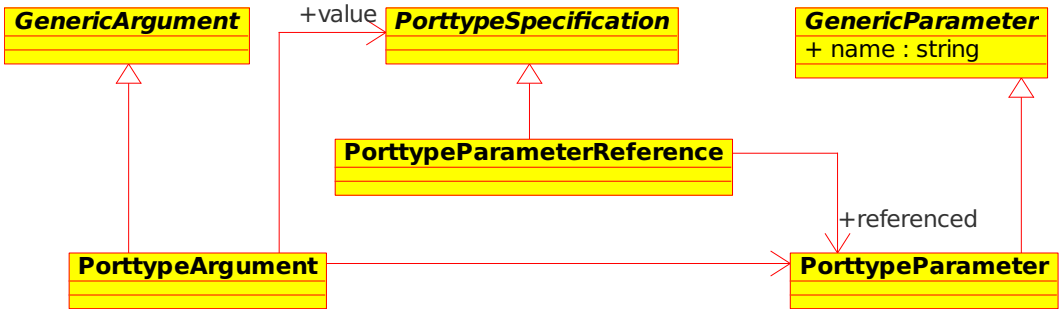

FIGURE 4.8 – Schéma UML de l'application du patron de conception pour permettre l'utilisation de type de ports comme paramètre

De la même manière, les méta-classes `GenericTypeDefinition` et `Specialization` doivent être spécialisées pour les types effectivement rendu génériques. Le schéma UML de la figure 4.9 présente un exemple où les types de composants sont rendu génériques. Dans ce cas, deux méta-classes sont créées : `GenericComponenttypeDefinition` et `ComponenttypeSpecialization` qui héritent de `GenericTypeDefinition` et `Specialization` respectivement. La méta-classe `GenericComponenttypeDefinition` contient en plus des paramètres qui lui viennent de `GenericTypeDefinition` un `ComponenttypeDefinition` qui représente le contenu du type générique. La méta-classe `ComponenttypeSpecialization` possède une référence vers un `GenericComponenttypeDefinition` et hérite de `ComponenttypeSpecification` pour modéliser le fait qu'une spécialisation peut être utilisée quand une spécification de type de composant est requise.

Valeurs par défaut L'ajout de la possibilité pour les paramètres de types génériques d'avoir des valeurs par défaut consiste simplement à leur permettre de contenir une spécification correspondant à la valeur par défaut. Dans l'exemple du schéma de la figure 4.10, la classe `PorttypeParameter` peut contenir une instance de la classe `PorttypeSpecification` qui représente sa valeur par défaut. Cette agrégation possède une multiplicité $0..1$ pour modéliser le fait que la spécification d'une valeur par défaut pour les paramètres est facultative.

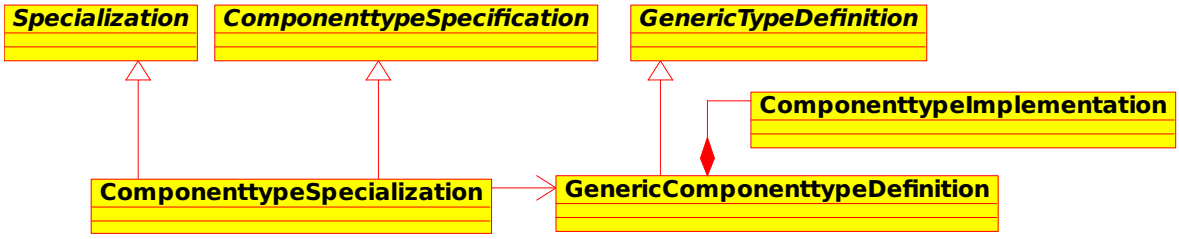

FIGURE 4.9 – Schéma UML de l'application du patron de conception pour rendre `ComponentType` générique

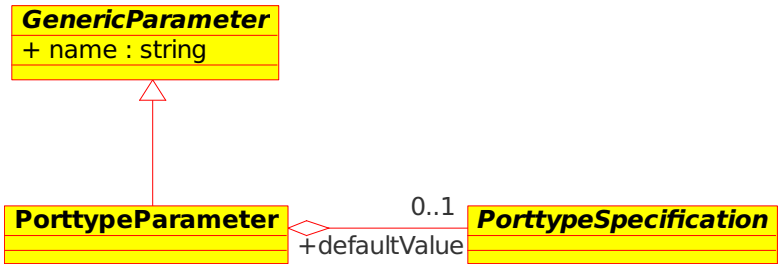

FIGURE 4.10 – Schéma UML de l'ajout du support d'une valeur par défaut aux paramètres de type `ComponentType` générique

Contraintes sur les paramètres Au moins deux approches peuvent être adoptées pour modéliser des contraintes sur les paramètres de types génériques. Une première approche consiste à attacher ces éventuelles contraintes sur chaque paramètres, comme c'est le cas pour les paramètres de génériques en JAVA par exemple. Une seconde approche consiste à les attacher au type générique lui même et à y faire référence aux paramètres.

Cette seconde approche a l'avantage de permettre d'exprimer des contraintes faisant intervenir plus d'un paramètre. Il est toutefois possible qu'elle rende plus compliqué l'analyse de ces contraintes pour en déduire les utilisations valides du type passé en paramètre. Dans la mesure où le choix a été fait précédemment de ne pas s'appuyer sur les contraintes pour déterminer les utilisations valides des types passé en paramètre, ce point ne pose pas problème. La modélisation des contraintes sera donc faite selon la seconde approche.

Une classe abstraite `Constraint` présentée sur le schéma de la figure 4.11 est utilisée pour modéliser les contraintes. La classe `GenericTypeDefinition` porte un ensemble de contraintes qui restreignent les valeurs valides pour ses paramètres.

Il existe de nombreuses contraintes possibles, et cet ensemble dépend du modèle de composants utilisé comme base pour l'ajout de la généricité. Une contrainte du type «le paramètre `X` représente une interface qui hérite de l'interface `Y`» n'a par exemple de signification que pour un modèle de composants comportant une notion d'héritages entre interfaces. L'ensemble des contraintes à modéliser dépend donc du modèle de composants cible, elles sont modélisées par des classes qui héritent de `Constraint`. Par exemple la classe `CompatibilityConstraint` modélise une contrainte de compatibilité entre plusieurs types de ports représentés par des instances de type dérivés de la classes `PorttypeSpecification`. Il peut notamment d'agir de la classe `PorttypeParameterReference` de la figure 4.8 auquel cas la contrainte s'applique à un paramètre.

Spécialisations explicites Modéliser les spécialisations explicites revient à permettre plusieurs mises en œuvres des types génériques en fonction de la valeur de contraintes données. Plusieurs possibilités peuvent à nouveau être choisies dans ce but. La première approche —proche de celle choisie pour le langage C++— consiste à permettre la description de plusieurs mises en œuvre complètement indépendantes, choisies en fonction de la va-

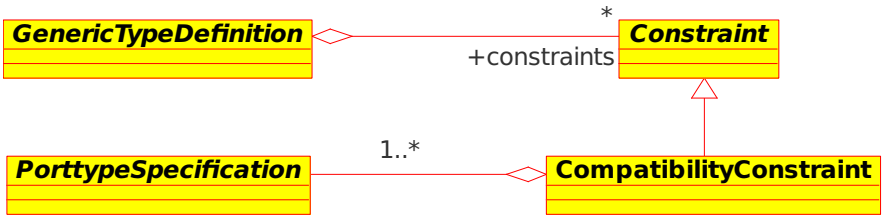

FIGURE 4.11 – Schéma UML de l'ajout du support de contraintes sur les paramètres de type générique

leurs de certaines contraintes. Une seconde approche consiste à proposer des structures de contrôle statique similaires au «*static if*» du langage D [1]. Ces structures qui permettent de modifier le contenu d'une partie d'une mise en œuvre donnée en fonction de la valeur d'une condition.

La première approche offre à l'utilisateur une vision plus proche des langages fonctionnels, bien adaptée au cas où les différentes mises en œuvre sont complètement indépendantes. La seconde approche offre une vision plus proche des langages impératifs et permet d'offrir une vision plus adaptée à la méta-programmation en ajoutant d'autres structures de contrôles telles que par exemple des boucles statiques.

Pour supporter la première approche une classe `ExplicitSpecialization` est utilisée pour modéliser les spécialisations explicites et pour chaque type générique une nouvelle classe qui en hérite est ajoutée, comme par exemple la classe `ComponenttypeExplicitSpecialization` présentée sur le schéma de la figure 4.12. La classe `ExplicitSpecialization` agrège une contrainte qui détermine quand elle doit être utilisée. Chaque classe qui en hérite est composé d'une définition du type qui représente la mise en œuvre a utiliser quand cette contrainte est remplie.

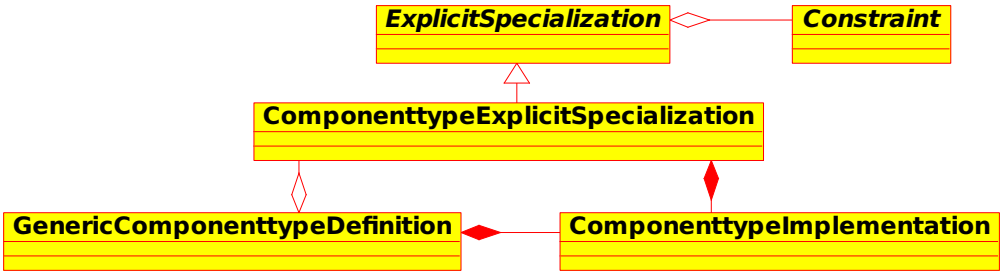

FIGURE 4.12 – Schéma UML de l'ajout du support des spécialisations explicites selon la première approche proposée

La seconde approche nécessite l'ajout dans le métamodèle de classes supportant les constructions de contrôle statique pour chaque type de mise en œuvre. Devant la grande hétérogénéité de ces mises en œuvre suivant le type, il est difficile de proposer une patron de conception toujours applicable. Le choix entre les deux approches doit donc se faire au cas par cas.

4.2.2 Support à l'exécution

L'algorithme présenté dans cette section prend en entrée un assemblage de composants décrit dans un métamodèle —noté $G(B)$ dans la suite— qui comporte de la généricité. Ce métamodèle $G(B)$ a été obtenu en appliquant le patron de conception de la section 4.2.1 à un métamodèle B. L'algorithme génère en sortie un assemblage de composant décrit dans ce métamodèle B qui ne comporte donc plus de généricité ou bien une erreur.

1. `http://www.digitalmars.com/d/2.0/version.html#staticif`

On considère qu'un assemblage (c'est à dire une application), aussi bien dans B que dans $G(B)$ est défini par un type de composant *a priori* composite et qui n'expose pas de ports. Il s'agit donc de produire étant donné un type de composant de $G(B)$ son équivalent dans B. Il faut aussi produire l'équivalent de tous les autres types dont la définition du premier dépend et ce récursivement.

La fonction principale de l'algorithme prend en entrée o, un objet du métamodèle et c, un contexte de transformation, c'est à dire une mise en relation des paramètres du type avec leur valeur. Le comportement de cette fonction de transformation dépend du type de o.

Si o représente un *type générique*, c'est à dire une instance d'un sous-type de `GenericTypeDefinition` de la figure 4.9, le contexte c est tout d'abord rempli avec les valeurs par défaut pour les paramètres de o qui n'y ont pas de valeur associée. Si après cette étape il reste des paramètres sans valeur dans c, l'algorithme termine par erreur. Les contraintes de o sont vérifiées et si l'une d'entre elle n'est pas remplie, l'algorithme termine par erreur. Si la première approche pour la modélisation des spécialisations explicites a été choisie, la fonction itère sur ces spécialisations explicites et choisie la première mise en œuvre pour laquelle les contraintes sont remplies. Sinon, la mise en œuvre par défaut est choisie. Quoi qu'il en soit, la fonction retourne un type non générique dont la mise en œuvre est obtenue par l'application de l'algorithme à la mise en œuvre choisie.

Si o représente une *référence à un paramètre*, comme par exemple une instance de la classe `PorttypeParameterReference` de la figure 4.8, la valeur associée à ce paramètre est recherchée dans c pour être retournée. Si sa valeur n'est pas trouvée, l'algorithme termine par erreur.

Si o représente une *spécialisation*, c'est à dire une instance d'un sous-type de `Specialization` de la figure 4.9, un nouveau contexte c' comportant les associations entre paramètres et valeurs spécifiées par les arguments de cette spécialisation est créé. Le résultat de la fonction consiste en l'application de l'algorithme à la définition de type générique référencée par o dans le nouveau contexte c'.

Si o n'*appartient à aucune des catégories précédentes*, une copie de o est créé pour être retournée. Chaque élément du contenu de o est remplacé dans cette copie par le résultat de l'application de l'algorithme à cet élément.

Une nécessité supplémentaire pour dire qu'une instance de $G(B)$ est valide est que l'application de cet algorithme génère une instance de B valide. Un exemple d'instance qui serait valide sans cette condition peut être constitué d'un type de composant composite contenant (éventuellement par transitivité) une instance de même type. Dans ce cas, l'algorithme ne terminerait pas et entrerait dans une récursion infinie. À l'inverse, si ce type est générique et accepte un paramètre dont la valeur change d'une itération à l'autre, il est possible qu'une spécialisation explicite finisse par être utilisée et arrête la récursivité.

En fait, comme nous avons précédemment remarqué que l'introduction de spécialisations explicites rend généralement le langage Turing-complet, le problème de la terminaison pour la compilation est *a priori* indécidable. Les compilateurs C++ qui font face au même problème imposent généralement une profondeur de récursion maximale au delà de laquelle une erreur est générée.

4.3 Validation

4.3.1 Application à SCA : GenericSCA

Le patron de conception présenté à la section 4.2.1 a été appliqué à SCA. Le métamodèle de SCA du «projet d'outils Eclipse pour SCA[2]» a été utilisé. Les concepts qui ont été rendu génériques dans le cadre de GenericSCA sont les composants (primitifs et composites). Le support des generics Java a aussi été ajouté. Les concepts qui peuvent être utilisés comme paramètres de type génériques dans GenericSCA sont les mises en œuvres de composants,

2. Eclipse SCA Tools projecthttp://www.eclipse.org/stp/sca/

les interfaces de ports, les types de données et les valeurs de données pour les composites. Les composants primitifs acceptent des types de données et les valeurs de données comme paramètres et les generics Java les classes Java.

Le métamodèle de GenericSCA est basé sur une copie de celui de SCA. Il comporte huit classes additionnelles pour modéliser la généricité : `GenericImplementation`, `ImplementationParameter`, `ImplementationArgument`, `GenericInterface`, `InterfaceParameter`, `InterfaceArgument`, `JavaTypeParameter` et `JavaTypeArgument`. Il n'a pas été nécessaire de créer une classe pour modéliser les types Java génériques car dans le métamodèle de SCA sur lequel nous nous sommes basé, les types Java sont simplement modélisés par une chaîne de caractères contenant leur nom. Il n'a pas non plus été nécessaire de modifier le métamodèle pour supporter les valeurs de données en paramètres car le concept de propriétés de configurations de SCA correspond déjà à cette utilisation.

Le support des contraintes pour les paramètres des types de composants génériques passe par l'ajout d'une classe au métamodèle : `Constraint` de laquelle dérivent toutes les autres. Les différentes classes de contraintes existantes dans GenericSCA sont les suivantes. Une classe `XpathConstraint` modélise des contraintes sur des valeurs de données exprimées sous la forme d'une expression XPath booléenne qui constitue la manière classique de manipuler des valeurs de données en SCA. Pour les autres types de paramètre la seule classe de contrainte existante pour l'instant est constituée de l'égalité modélisée par les classes `ImplementationEqConstraint`, `InterfaceEqConstraint` et `JavaTypeEqConstraint`. Finalement, trois classes de contraintes servent à modéliser les combinaisons logiques d'autres contraintes : `ConjonctionConstraint`, `DisjunctionConstraint` et `NegationConstraint`.

Le support des spécialisations explicites a été mis en œuvre selon la première approche proposée pour les composants composites. Le choix de cette approche est dû au fait qu'elle est moins invasive et donc plus facile à mettre en œuvre dans le cadre d'une modification d'un métamodèle existant. Une classe `compositeSpecialization` a été ajoutée au modèle qui modélise cet aspect.

4.3.2 Mise en œuvre

Un prototype de mise en œuvre de l'algorithme de transformation d'assemblages GenericSCA en assemblages SCA de base a été réalisé et est disponible au téléchargement (`http://graal.ens-lyon.fr/~jbigot/genericSCA`). Il met en œuvre l'algorithme présenté en 4.2.2. Le cas des generics Java est particulier puisqu'il consiste simplement à vérifier la compatibilité des types utilisés comme paramètre et à les effacer. En effet, Java gère les generics à l'exécution par effacement de type, les paramètres ne sont utilisés que pour assurer la sûreté de typage à la compilation et sont complètement oubliés à l'exécution.

Puisque les métamodèles manipulés sont modélisés en Ecore, il a tout d'abord semblé intéressant de s'appuyer sur les langages spécifiques au domaine (*Domain Specific Language* – DSL) dédiés à la transformation de modèle dans EMF. Un premier essai a été effectué avec QVT (*Query View Transform*) et plus particulièrement sa version impérative QVTO (QVT *Operational*). Le support de ce langage n'était toutefois pas satisfaisant lors du développement du prototype et l'algorithme de transformation a finalement été développé en Java.

Les classes Java qui correspondent à celles du métamodèle de GenericSCA ont été automatiquement générées. Le parseur pour instancier ces classes depuis les documents XML SCA a aussi été automatiquement généré grâce à des annotation du métamodèle de SCA. Les classes Java correspondant au métamodèle de SCA et le code utilisé pour générer les documents XML SCA une fois la transformation effectuée sont ceux du projet d'outils Eclipse pour SCA. Finalement, plus de 50 000 lignes de code Java ont été générées et autant sont issues de la réutilisation de code existant.

La mise en œuvre du cœur de l'algorithme de transformation requiert environs 750 lignes de Java dont la majorité consiste simplement en des copies d'attributs de classes GenericSCA vers l'attribut du même nom d'une classe SCA (dernier cas de l'algorithme de la section 4.2.2). Ce code aurait pu être généré si un langage dédié à la transformation de modèle comme QVT avait pu être utilisé. La logique de l'algorithme elle-même ne constitue

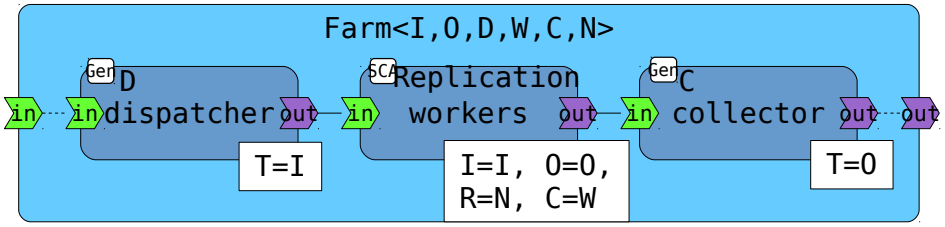

FIGURE 4.13 – Le composant générique `Farm`. Il s'agit d'un composite qui contient trois instances `dispatcher`, `workers` et `collector`.

qu'environs 100 lignes de ce code bien qu'il soit difficile de donner une estimation précise puisqu'elle est fortement imbriqué avec le reste.

4.3.3 Exemples d'utilisation de GENERICSCA

Ce prototype a été validé en développant en GENERICSCA une mise en œuvre du squelette ferme de tâches. Il s'agit d'un composant composite générique acceptant en paramètres des types java, des mises en œuvre de composant et une valeur de donnée. Sa mise en œuvre repose sur de la méta-programmation pour contenir le bon nombre d'instances de composant. Cette mise en œuvre de la ferme de tâches a été utilisée avec succès pour calculer des images de l'ensemble de Mandelbrot pixel par pixels et par blocs d'image. La fin de cette section présente cet exemple plus en détail.

Le composant générique «Farm» Le composant générique `Farm` présenté graphiquement sur la figure 4.13 est un composite qui accepte six paramètres :

- deux paramètres sont des types JAVA (`I` et `O`) qui décrivent le type des données en entrée et sortie de la ferme respectivement ;
- trois paramètres décrivent des mises en œuvres de composants (`D`, `W` et `C`) qui définissent les types du distributeur de données, des travailleurs et du collecteur respectivement ;
- un paramètre est un entier (`N`) qui définit le nombre de travailleur dans la ferme.

Ce composite contient simplement une instance de chacun des composants `D` et `C`. Il s'appuie sur le composite `Replication` décrit plus en détail ci-après pour créer des instances du composant `W`. Ces instances sont connectés par un flux de donnée mis en œuvre grâce à l'interface JAVA générique `DataPush<A>` qui comporte une unique méthode `void push(A data)`. Cette interface est utilisée avec `T` comme argument avant le passage dans les travailleurs et avec `O` après.

Les paramètres `D` et `C` ont des valeurs par défaut : `RRDispatcher<T>` et `SimpleCollector<T>` respectivement. Ces composants distribuent les données selon l'algorithme du tourniquet (*round-robbin*) et les collectent sans réordonnancement particulier. Ce sont des mises en œuvres JAVA génériques qui ne dépendant pas du type des données manipulées.

Le composant générique «Replication» Le composant générique `Replication` présenté graphiquement sur la figure 4.14 réplique un certain nombre d'instances d'un type de composant donné. Il accepte quatre paramètres :

- deux paramètres sont des types JAVA (`I` et `O`) qui définissent le type des données en entrée et en sortie du composant ;
- un paramètre est une mise en œuvre de composant (`C`) qui définit le type du composant répliqué ;
- le dernier paramètre est un entier (`R`) qui définit le nombre de fois que le composant est répliqué.

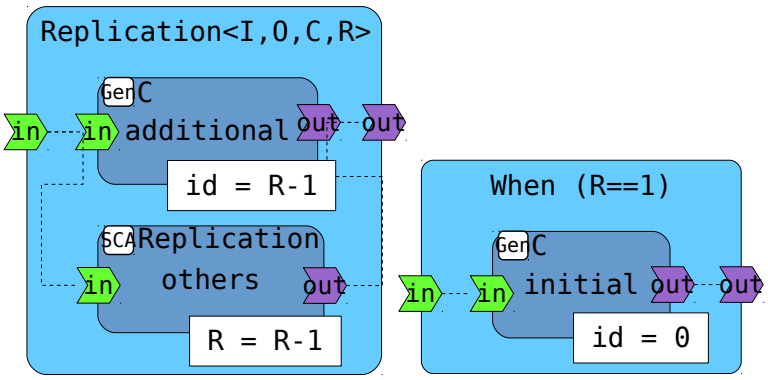

FIGURE 4.14 – Le composant générique `Replication`. Il s'agit d'un composite qui contient deux instances `additional` et `others`. Il possède une spécialisation explicite utilisée quand la valeur du paramètre `R` est 1. Cette spécialisation explicite ne contient qu'une instance `initial`.

Sa mise en œuvre s'appuie sur la méta-programmation et sur la récursivité. Elle contient une instance de `C` appelée `additional` et une instance de `Replication` où la valeur du paramètre `R` est décrémentée de 1. Le cas de base de la récursivité est définit par la spécialisation explicite utilisée quand la valeur de `R` atteint 1. Dans ce cas, seule l'instance de `C` est créée.

Évaluation Cet exemple démontre qu'il est possible de mettre en œuvre des squelettes algorithmiques à l'aide des concepts de généricité et de hiérarchie. L'introduction de la méta-programmation rend cependant l'analyse du comportement de tels composants complexe à analyser.

Cette mise en œuvre de la ferme de tâches a été utilisée pour calculer des images de l'ensemble de Mandelbrot. Deux sortes de travailleurs ont été développés et utilisés au sein de la ferme :

- le premier calcule la valeur d'un unique pixel à la fois ;
- le second travaille sur des morceaux plus importants de l'image dont la taille exacte peut être paramétrée.

Ces deux versions ont été déployées avec succès au sein de la ferme sur une machine multi-cœur en faisant varier le nombre de travailleurs entre un et quatre. La phase de transformation a nécessité entre une et deux secondes avec l'essentiel du temps utilisé à analyser les fichiers en entrée.

4.4 Conclusion

Cette section a étudié la faisabilité d'introduire la généricité dans les modèles de composants afin de pouvoir y décrire de nouveaux squelettes. Pour cela, un patron de conception qui permet d'étendre un modèle de composants existant pour lui ajouter la généricité a été proposé. Un algorithme qui étant donné un assemblage décrit dans ce nouveau nouveau modèle permet de générer un assemblage équivalent dans le modèle d'origine a aussi été décrit. Ceci a été appliqué à SCA pour donner GENERICSCA qui a été validé en mettant en œuvre une application de rendu d'image basé sur le squelette de ferme de tâches.

CHAPITRE 5

Introduction du concept de connecteur dans les modèles de composants hiérarchiques

Le chapitre 3 a présenté l'approche que nous proposons pour concevoir un modèle de composants extensible par des fonctionnalités dédiées au calcul à haute performance. Cette approche est caractérisée par la description des applications dans un modèle qui offre un haut niveau d'abstraction et par une transformation vers un modèle plus proche du matériel lorsque les ressources d'exécution sont connues. Un des besoins identifiés pour le modèle de haut niveau est la possibilité d'y décrire les interactions entre composants de manière abstraite de manière à pouvoir en proposer plusieurs mises en œuvres, chacune destinée à des ressources d'exécution différentes.

Cette possibilité constitue le principal attrait des modèles de composants qui incluent le concept de connecteur comme entité de première classe. Ce chapitre évalue donc la possibilité d'ajouter ce concept à un modèle qui comporte l'ensemble des fonctionnalités décrites dans les chapitres précédents. Il analyse notamment l'interaction entre les concepts de connecteur et de hiérarchie.

Cette analyse montre que l'approche habituellement adoptée présente une lacune qui dans certain cas empêche de concilier performances optimales et interchangeabilité des mises en œuvres de composant. Une approche est proposée qui résout ce problème en introduisant une nouvelle description l'interface des composants à l'aide du concept nouveau de connexion ouverte dont le polymorphisme est supporté par le concept nouveau d'adaptateur de connexion. Le concept de port *bundle* est aussi introduit pour permettre une meilleure mise en œuvre de certaines catégories de connexions.

La section 5.1 introduit des exemples synthétiques d'applications et les utilise pour évaluer l'approche adoptée dans les modèles de composants existant pour supporter les connecteurs. La section 5.2 présente notre approche qui s'appuie sur les concepts de connexion ouverte, d'adaptateur de connexion et de *bundles*. Cette approche est utilisée pour ajouter le support des connecteurs à un modèle de composants générique hiérarchique pour former le modèle appelé *High Level Component Model* (HLCM). Finalement la section 5.3 s'appuie sur une spécialisation de HLCM (HLCM/CCM) pour évaluer l'approche et la section 5.4 conclue brièvement.

5.1 Analyse préliminaire

Parmi les propriétés identifiés comme nécessaire au chapitre 3 figure la possibilité de choisir la mise en œuvre des interactions entre composants en fonction des ressources d'exécution sur lesquelles ils sont déployés. Le concept de connecteur en tant qu'entité de première classe qui a été introduit dans certains modèles de composants vise notamment à assurer cette propriété. Cette section évalue l'approche adoptée dans ces modèles pour introduire le concept de connecteur et son applicabilité à un modèle qui comporte l'ensemble des fonctionnalités décrites dans les chapitres précédents.

Dans un premier temps, les modèles de composants qui supportent le concept de connecteur comme entité de première classe sont présentés et l'approche qu'ils adoptent pour l'introduire est analysée. Ensuite, des exemples synthétiques d'applications sont introduits et sont décrits en s'appuyant sur des connecteurs introduits selon cette approche. Finalement, l'approche est évaluée en s'attardant sur sa capacité à supporter l'interchangeabilité des mises en œuvres de composant.

5.1.1 Présentation de l'existant

Un connecteur est un type destiné à supporter la spécification d'interactions entre les composants. Ces interactions sont exprimés par des connexions qui sont des instances de connecteur. Le concept de connecteur trouve son origine dans les langage de description d'architecture (*Architecture Description Languages* – ADL) mais il a été intégré au sein de modèles de composants dans le modèle SOFA [11] ainsi que dans un modèle proposé par Matougui et Beugnard [43]. Dans ces deux modèles, un connecteur est une entité de première classe : de nouveaux connecteurs ainsi que de nouvelles mises en œuvres des connecteurs existant peuvent être définis par l'utilisateur.

Sofa L'introduction du concept de connecteur dans SOFA est motivée par la volonté de supporter la distribution indépendamment de l'intergiciel utilisé. Il s'agit notamment de permettre d'utiliser diverses mises en œuvres d'un type spécifique d'interaction (l'appel de méthode) sans nécessiter de modification des composants. Dans ce modèle, le concept que nous appelons connecteurs est nommé *connector template* et ce sont les instances de *connector templates* (que nous appelons connexions) qui y sont appelées connecteurs. Chaque *connector template* et formée d'une *frame* qui constitue son interface et d'une *architecture* qui forme sa mise en œuvre. Les *connector template* peuvent être paramétrés par des interfaces objets et par des attributs.

La *frame* d'un *connector template* est formée d'un ensemble de *rôles* `provides` ou `requires` typés par une interface objet. Dans les instances de *connector template*, chaque *rôle* est rempli par un port du même type mais de la classe complémentaire (un *rôle* `provides` et rempli par un port `requires` et inversement). Quand l'interface qui type un *rôle* est un paramètre du *connector template*, le paramètre prend automatiquement la valeur apportée par le port qui remplit le *rôle*.

Il existe deux sortes d'*architectures* de *connector template*. Les *architectures composées* (*compound*) sont constituées d'un assemblage à la manière des mises en œuvres composites de composant. Dans cet assemblage, les ports qui remplissent les *rôles* du connec-

teur peuvent être utilisés pour remplir des *rôles* de connecteurs internes à l'assemblage. Les *architectures simples* sont constituées d'*éléments primitifs*, c'est à dire de code tel que les souches (*stubs*) et les squelettes (*skeletons*) dans le cas d'un connecteur CORBA.

Modèle de Matougui et Beugnard L'introduction du concept de connecteur dans le modèle proposé par Matougui et Beugnard répond à deux objectifs : permettre la description d'interactions complexes et supporter la distribution de manière transparente. Ces deux objectifs donnent lieu à deux abstractions distinctes : les composants de communication pour décrire des interactions complexes et les connecteurs pour supporter la distribution.

Un composant de communication est en tous points similaire aux composants classiques : il expose des méthodes qu'il fournit et d'autres qu'il requiert et peut être mis en œuvre de la même manière que les autres composants. La seule différence avec un composant classique réside dans sa sémantique : il ne décrit pas une partie d'une application mais une interaction entre de telles parties.

Les connecteurs forment une abstraction légèrement différente : leur interface est formée d'un ensemble de *plugs* et leurs mises en œuvres —appelées générateurs— sont spécifiées indépendamment du connecteur. Le concept de *plug* est similaires à celui de port à la différence près que l'interface qu'il supporte n'est pas spécifiée et constitue un paramètre générique du connecteur. Les *plugs* sont nommés et peuvent spécifier une multiplicité. Les instances de connecteurs sont appelées connexions.

Les générateurs mettent en œuvre les connecteurs en générant du code. Cette opération de génération peut s'appuyer sur le type des ports de composants connectés aux *plugs* de la connexion mise en œuvre. Le code résultant de la génération est identifié sous le nom de *binding component*.

Discussion Les deux modèles présentés adoptent des approches très similaires pour l'introduction du concept de connecteur dans les modèles de composants. Les ports exposés par les composants ne sont plus directement connectés mais interagissent en participant à une même connexion. Les connexions sont des instances de connecteur (ou *connector template* dans le cas de SOFA) qui possèdent un ensemble de *rôles* (ou *plugs* dans le cas du second modèle). Ce sont ces *rôles* qui sont remplis par les ports des composants. Dans le premier modèle, chaque *rôle* ne peut être rempli que par un unique port alors que le second modèle supporte une multiplicité associée à ces *rôles*.

Les connecteurs peuvent posséder plusieurs mises en œuvres et le type des ports qui participent aux connexions constituent des arguments génériques qui peuvent être utilisés dans ces mises en œuvres. Il existe deux manières de décrire ces mises en œuvres :

- elles peuvent être constituées d'un système de génération de code qui produit par exemple les souches et les squelettes habituellement utilisés pour supporter l'appel de méthode à distance ; ou
- il peut s'agir d'un assemblage à la manière des composites, bien que dans ce cas le second modèle étudié parle de composant de communication et ne permette pas d'utiliser le type des ports des composants connectés comme paramètres génériques

Il faut cependant noter que ces deux types de mises en œuvre s'appuient sur des interactions de plus bas niveau et que l'interaction par appel local de méthode est nativement supporté pour amorcer le processus.

Cette description des connecteurs est très similaires à celle des composants dans le modèle générique et hiérarchique que nous avons construit aux chapitres 3 et 4. Comme ces composants, les connecteur acceptent des paramètres génériques qui peuvent notamment être utilisés pour typer leurs points d'interaction. Comme ces composants, ils peuvent avoir plusieurs mises en œuvres dont une sera choisie au déploiement. Comme pour ces composants, leurs mises en œuvres peuvent être formées par un assemblage ou être primitives.

Il existe cependant des différences qui justifient l'introduction d'un concept distinct. Les valeurs des paramètres génériques d'un connecteur sont déduites des ports qui remplissent ses *rôles* alors que ceux d'un composants doivent être explicitement spécifiés. Dans un

même modèle, la manière de décrire les mises en œuvres primitives des composants et des connecteurs diffère. Finalement, la sémantique supportée par chacun des concepts diffère.

Une première solution pour introduire le concept de connecteur dans un modèle de composants qui correspond à ce qui a été décrit dans les chapitres 3 et 4 consiste à adopter cette approche. C'est donc ce que nous proposons de faire afin d'évaluer la viabilité d'une telle approche dans le reste de cette section.

Le modèle évalué comporte des connecteurs qui sont similaires aux composants sauf qu'ils exposent des *rôles* au lieu de ports qui sont implicitement des paramètres génériques du connecteur. Les ports des composants ne peuvent pas être connectés entre eux mais doivent remplir les *rôles* de connexions qui sont des instances de connecteurs. Ces connexions sont instanciées dans l'assemblage de la même manière que les connecteurs.

Chaque *rôle* d'un connexion est rempli par un ensemble de ports sans contrainte de nombre ou de types mais les mises en œuvres des connecteurs appelées générateurs peuvent imposer des contraintes. Il existe deux sortes de générateurs : les générateurs primitifs supportés par le modèle sous-jacent et les générateurs composites. Un générateur composite est un assemblage dans lequel les ports qui remplissent ses *rôles* peuvent être utilisés pour remplir des *rôles* de connexion internes.

5.1.2 Exemples synthétiques d'application

Pour évaluer ce modèle, on s'appuie sur deux variations autour d'un exemple synthétique d'application qui sont présentées ici. Elles sont conçues pour être représentatives d'applications de couplage de code [52] telles qu'on les trouve par exemple dans le domaine du rendu d'image [7], de l'hydrologie [19] ou de la dynamique moléculaire quantique [14].

Ces applications sont formées par le couplage de codes développés par des équipes indépendantes qui possèdent chacune une expertise d'un domaine différent. Ces codes peuvent être séquentiels mais ils sont souvent parallèles, de type SPMD. Dans ce dernier cas, il s'agit d'un ensemble de processus dont le nombre peut être adapté aux ressources d'exécution et qui communiquent entre eux par passage de message et notamment à l'aide d'opérations de communication collective. Le choix de la version la plus adaptée des codes ainsi que de paramètres tels que le nombre de processus dans le cas de la version SPMD doit être fait en fonction des ressources d'exécution sur lesquelles ils sont déployés.

Pour étudier ce type d'application, on s'intéressera au cas du couplage de deux codes qui possèdent chacun deux versions :

- une version séquentielle ; et
- une version parallèle pour laquelle le degré de parallélisme est un paramètre.

On s'intéressera à deux types d'interactions différentes pour le couplage entre ces deux codes.

Dans la première version, les codes interagissent par des appels de méthodes. Ce type d'interaction est représentatif d'applications [19, 14] où les codent alternent entre phases de calcul où un pas de temps est simulé et phases de communication où les modifications de l'état global de l'objet simulé sont échangées. Cet échange d'information est porté par les paramètres de méthodes appelée entre les différents codes qui forment l'application.

Dans le cas où les versions séquentielles des codes sont utilisés, l'interaction est un appel de méthode classique. Si ces codes sont déployés sur des ressources d'exécution différentes, il s'agit toutefois d'un appel de méthode à distance. Quand la version parallèle de l'un des codes est utilisée, l'interaction devient un appel de méthodes parallèle et dans le cas où les deux codes sont parallèles il s'agit du cas d'un appel de méthodes parallèle $M \times N$.

Dans une seconde version, les codes interagissent en partageant logiquement un espace de mémoire commun. Ce type d'interaction est représentatif d'applications [7] dans lesquelles les codes ne suivent pas de motif régulier et peuvent entraîner des modifications de parties difficiles à prévoir de l'état global. Cet état est généralement stocké dans une mémoire accessible par l'ensemble des codes qui utilisent des mécanismes de verrou pour assurer l'intégrité des données qui y sont stockées.

Ici, le choix des versions séquentielles ou parallèles des codes a peu d'influence sur le choix de la mise en œuvre de l'interaction. Dans tous les cas, un certain nombre de processus accèdent à une mémoire logiquement partagée et le fait qu'ils participent logiquement à un même code ou non influe peu. L'aspect qui est important est le nombre de processus et leur distribution sur les ressources d'exécution. Les mises en œuvres adaptées à l'échelle d'une machine à mémoire physiquement partagée et celles adaptées à l'échelle de la grille de calcul sont en effet très différentes.

Description à base de composants Ce type d'applications se prête tout particulièrement à un développement basé sur des composants. En effet, il s'agit de codes relativement indépendants et développés par des équipes différentes. L'utilisation de composants facilite le couplage de ces codes en permettant d'identifier clairement leurs points d'interaction. Afin de profiter de cet avantage, chaque version de l'exemple est décrite sous la forme d'un assemblage de deux composants qui encapsulent chacun l'un des codes couplés.

Chacun de ces codes existe en deux versions : l'une séquentielle, l'autre parallèle. Afin de pouvoir optimiser l'application pour les ressources d'exécution disponibles, il convient de permettre le choix entre les différentes versions de chaque code sans nécessiter de modification de la description de l'application. C'est dans ce but qu'a été introduite la possibilité de décrire de multiples mises en œuvres des composants. Les différentes versions de chaque code sont donc décrites comme des mises en œuvres distinctes d'un unique composant.

Les versions séquentielles des codes peuvent être décrites sous la forme de mises en œuvres primitives de composants ou bien elles peuvent s'appuyer sur des composants de grain plus fin regroupés au sein d'une mise en œuvre composite. Les versions parallèles pourront typiquement être mises en œuvre en utilisant un squelette de composant SPMD parallèle décrit à l'aide de composites et de la généricité comme présenté au chapitre 4.

Au sein de ces assemblages, il existe trois catégories d'interactions :

- à l'intérieur des composants parallèles, les instances de composant qui supportent le parallélisme interagissent à l'aide de passage de message et de communication collective ;
- dans la première version de l'exemple, les deux instances de composant qui forment l'application interagissent par appel de méthode ; et
- dans la seconde version de l'exemple, les deux instances de composant qui forment l'application interagissent par partage de mémoire.

Chacune de ces interactions logiques peut être mises en œuvre de différentes manières. Le choix de la mise en œuvre la plus adaptée dépendant de la mise en œuvre des composants qui participent à l'interaction et des ressources d'exécution sur lesquelles ils sont déployés. Ces caractéristiques caractérisent une situation où l'utilisation de connecteurs pour décrire les interactions est intéressante.

Utilisation des connecteurs Une première catégorie d'interaction utilisée dans ces exemple est constituée par l'appel de méthode. Cette interaction implique deux participants et peut être supportée par un connecteur avec deux *rôles*. Un *rôle* `user` correspond à l'utilisateur du service et un *rôle* `provider` à son fournisseur du service.

Suivant le modèle sous-jacent, les types de ports qui peuvent participer à une telle interaction peuvent varier. On s'intéressera à un modèle qui supporte classiquement des types de port primitifs `Uses<T>` et `Provides<T>` où `T` est une interface objet. On peut aussi considérer que deux générateurs primitifs existent qui mettent en œuvre le connecteur quand le *rôle* `user` est rempli par un unique port de type `Uses<U>` et que le *rôle* `provider` est rempli par un unique port de type `Protides<P>` et que `P` hérite de `U`. Le premier met en œuvre le connecteur lorsque les deux composants impliqués sont localisés au sein d'un même processus alors que le second gère l'appel distant.

Le cas des appels de méthodes parallèles au contraire doit être mis en œuvre par des générateurs décrits par l'utilisateur. Il est nécessaire de décrire trois générateurs pour gérer les cas 1 vers N, N vers 1 et M vers N. Dans l'exemple du générateur qui supporte

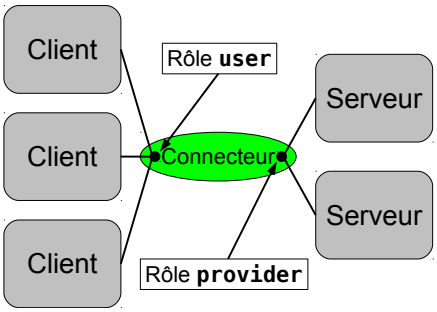

FIGURE 5.1 – Utilisation d'un connecteur pour décrire une interaction par appel de méthode parallèle. Le connecteur représenté par une ellipse possède deux *rôles* représentés par des cercles pleins : le *rôle* `user` et le *rôle* `provider`. Chacun de ces *rôles* est rempli les ports de plusieurs instances de composant pour gérer le cas de l'appel de méthode $M \times N$ dans le cas représenté de deux codes parallèles de degré 3 et 2.

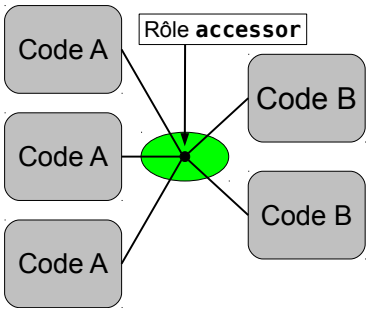

FIGURE 5.2 – Utilisation d'un connecteur pour décrire une interaction par partage de mémoire. Le connecteur possède un un unique *rôle* `accessor` repli par les ports de tous les composants qui accèdent à la mémoire dans le cas représenté de deux codes parallèles de degré 3 et 2.

le cas de M vers N, les *rôles* `user` et `provider` doivent être remplis par plusieurs ports de type `Uses<Part<U> >` et `Provides<Part<P> >` respectivement où `Part<P>` est une interface générique dans laquelle seule une partie de chaque argument est passée. Un exemple d'utilisation de ce connecteur est présenté sur le schéma de la figure 5.1.

La seconde catégorie d'interaction utilisée est constituée du partage de mémoire. Cette interaction implique un nombre quelconque de participants sans qu'on puisse distinguer de rôle spécifique pour ces participants. Elle peut donc être supportée par un connecteur ne comportant qu'un seul *rôle* : `accessor`.

Ce connecteur peut être mis en œuvre par des générateurs qui nécessitent que ce *rôle* soit rempli par des ports de type `Uses<MemAccess>` où `MemAccess` est une interface qui permet l'accès à la mémoire et la manipulation de verrous pour l'accès concurrent. Ces différents générateurs peuvent imposer des contraintes différentes concernant la localisation des composants qui participent à l'interaction. Un exemple d'utilisation de ce connecteur est présenté sur le schéma de la figure 5.2.

Une troisième catégorie d'interaction est constituée du passage de message à la MPI. Il s'agit d'un cas très similaire à celui du partage de mémoire qui peut être supporté par un connecteur comportant un unique *rôle*.

Il faut cependant prendre en compte le fait que les composants sont encapsulés dans des composites. Dans le cas du partage de mémoire entre composants parallèles par exemple, en instanciant une connexion dans l'assemblage où sont instanciés les composites, on obtient un assemblage qui ressemble à celui présenté sur le schéma de la figure 5.3. Les composites doivent exposer les ports de leurs instances de composant internes pour que ces instances puissent participer à la connexion.

Cette description du composite ne décrit pas toute la sémantique requise. Le fait que les différentes instances de composant qui composent le composite doivent interagir par une mémoire partagée n'y est notamment pas spécifié. L'interface du composant est donc fragile et un utilisateur qui ferait le choix de ne pas connecter ces ports à une même connexion empêcherait le composant de fonctionner correctement.

De plus, cette approche expose la manière dont le composant est mis en œuvre. L'interface de la version parallèle présentée est ainsi différente de celle d'une version séquentielle qui n'exposerait qu'un unique port. En nécessitant des interfaces différentes, ces mises en œuvres ne peuvent donc pas constituer des mises en œuvres interchangeables d'un unique

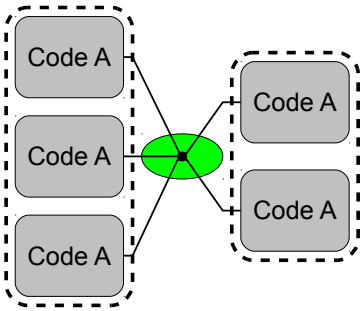

FIGURE 5.3 – Couplage de deux composants parallèles avec un connecteur instancié entre les composites. Les composites exposent les ports de toutes leurs instances de composant internes pour qu'elles puissent participer à la connexion.

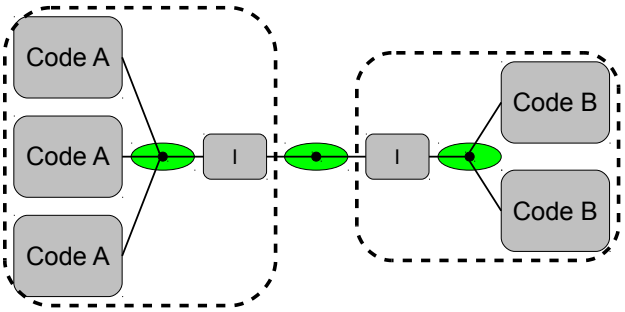

FIGURE 5.4 – Couplage de deux composants parallèles en instanciant des connecteurs entre les composites et au sein de chaque composite. Des composants sont introduits pour relier les connexions qui constituent un goulot d'étranglement possible.

composant.

Afin de contourner ce problème, il est nécessaire d'insérer à l'intérieur des composites une connexion qui exprime le fait que les instances de composants qui les composent interagissent. Toutefois, puisque les composants doivent exposer des ports, il devient alors nécessaire d'insérer en plus une instance de composant responsables de l'adaptation d'interface comme présenté sur le schéma de la figure 5.4.

Dans le cas du partage de mémoire par exemple, ce composant doit assurer que les mémoires partagées au sein des deux connexions auxquelles il participe sont toujours synchronisées. Ceci peut s'avérer très complexe dans le cas où les connecteurs n'ont pas été prévus pour ça. De plus, toutes les interactions entre les deux composites transitent par cette unique instance de composant qui peut alors former un goulet d'étranglement. Il s'agit d'un problème important pour les performances dans le cas de haut degrés de parallélisme comme on en trouve dans les applications de calcul à haute performance.

5.1.3 Discussion

Cette section a introduit deux exemples d'application qui ont été utilisés pour évaluer l'approche utilisée pour introduire les connecteurs dans dans les modèles de composants qui supportent ce concept. Cette approche permet de décrire des interactions qui impliquent plus de deux participants telles que l'appel de méthode parallèle, les communications collectives ou le partage d'une zone de mémoire. Elle permet de définir plusieurs mises en œuvres pour ces interactions parmi lesquelles un choix est effectué en fonction du placement des composants sur les ressources d'exécution.

Dans le cadre de modèles de composants hiérarchiques, il est possible que les composants primitifs qui participent aux interactions décrites par des connecteurs soient répartis au sein de plusieurs composites. Dans ce cas, la question se pose de la manière dont le connecteur doit être instancié.

En exposant les ports des instances de composant primitifs qui les composent, les composites affaiblissent leur encapsulation. Cela fragilise leur interface et empêche la description de mises en œuvres interchangeables pour les composants. En décrivant plusieurs connexions reliées par des composants les performances sont impactées, ce qui n'est pas acceptable dans le cas d'applications de calcul à haute performance.

La difficulté vient du fait que les interactions de haut niveau représentées par les connecteurs et la hiérarchie décrite par les composites forment deux regroupement orthogonaux des composants primitifs. L'exposition par les composites de tous les ports des instances de composant qui les composent n'est pas acceptable car elle rompt la logique de la hiérarchie

en exposant le contenu des composites. Inversement, l'insertion de composants pour relier des connexions qui se situent à différents niveaux de la hiérarchie n'est pas acceptable car elle impose un partitionnement des interactions logiques qui n'est pas guidée par le choix de la mise en œuvre la plus efficace. La section suivante propose une nouvelle approche pour l'utilisation des connecteurs qui ne présente pas cette limitation.

5.2 Un modèle de composants abstrait avec connecteurs

L'analyse présentée dans la section précédente montre que pour atteindre les objectifs fixés au chapitre 3, il est nécessaire de définir un modèle de composants dans lequel les connexions peuvent s'affranchir des barrières de la hiérarchie. Cette section propose dans ce but une nouvelle approche basée sur l'introduction de deux nouveaux concepts pour la description des applications : les connections ouvertes et les adaptateurs de connexions. Ces concepts sont supportés par une transformation de l'assemblage lors du déploiement de l'application.

Dans un premier temps, l'approche proposée est présenté et les choix effectués discutés. Ensuite, un patron de conception pour la modélisation des concepts requis est défini. Finalement, la transformation permettant le support à l'exécution de ces nouveaux concepts est spécifiée.

5.2.1 Présentation de l'approche

L'approche proposée dans cette section vise à concilier les besoins liés à la description de l'application et ceux liés à son exécution :

- dans la description des composants, une interaction possible doit être décrite par une unique entité dans l'interface des composants afin d'assurer une relative sécurité d'utilisation de cette interface ;
- l'entité exposée dans l'interface doit pouvoir être identique pour deux mises en œuvres différentes tant qu'elles sont compatibles en terme de connexion pour permettre l'interchangeabilité des mises en œuvres de composants ; et
- au moment du choix de leur mise en œuvre, les instances des connecteurs doivent relier directement les primitifs impliqués afin de pouvoir choisir celle la plus adaptée.

L'approche proposée pour concilier ces besoins consiste à appliquer une transformation qui génère un assemblage dans lequel les connexions relient directement les composants primitifs à partir d'un assemblage hiérarchique où chaque interaction n'est décrite que par une seule entité et où les mises en œuvres sont interchangeables.

Connexions ouvertes et fusion La première difficulté identifiée concerne les composites qui contiennent une connexion à laquelle plusieurs instances de composants internes participent tout en nécessitant la participation d'instances externes au composite. Pour décrire le fait que des instances internes au composite interagissent, la seule solution consiste à instancier un connecteur à l'intérieur du composite. Cependant, si les points d'interaction des composants sont définis par des ports on se retrouve dans la situation présentée sur la figure 5.4 où des artefacts qui ne portent aucune sémantique morcellent les connexions. Il est donc nécessaire de proposer un nouvelle approche pour décrire l'interface des composites.

Une approche pourrait consister à exposer des *rôles* de la connexion interne pour qu'ils puissent être remplis de l'extérieur comme illustré sur le schéma de la figure 5.5 pour l'exemple de l'interaction par partage de mémoire entre deux composants parallèles. Il faut cependant noter qu'il existe plusieurs *rôles* par connexion, ce qui ne répond pas au besoins de n'exposer qu'une entité par interaction logique. De plus, comme on le voit sur le schéma, l'interaction entre deux composants qui exposent un *rôle* chacun nécessiterait l'insertion d'un composant intermédiaire puisque des *rôles* ne peuvent pas être connectés entre eux. Ce point s'oppose au besoin d'obtenir des connexions directes sans insertion d'artefacts sans

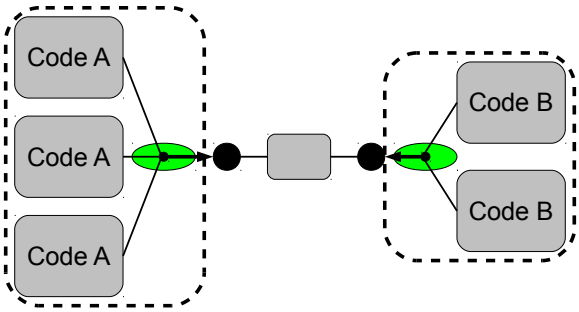

FIGURE 5.5 – Exemple du partage de mémoire entre deux composants parallèles dans le cas des interfaces de composants décrites par l'exposition de *rôles*. Chaque composite expose un *rôle* et il est nécessaire d'insérer une instance de composant pour faire l'adaptation entre les deux composites car deux *rôles* ne peuvent pas être directement connectés.

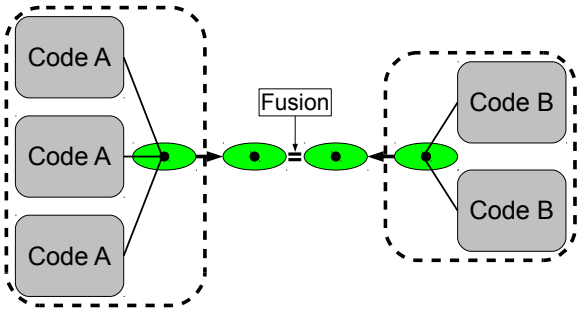

FIGURE 5.6 – Fusion des connexions exposées par deux composites pour ne plus former qu'une unique connexion logique.

sémantique dans l'assemblage. Pour résumer, cette approche pose les mêmes problèmes que l'approche où ce sont les ports qui sont exposés.

En fait, ces deux approches se caractérisent par une dissymétrie entre *rôles* et ports bien adaptée au cas des interactions point à point comme l'appel de méthodes mais qui pose problème dans le cas d'un interaction symétrique comme le partage de mémoire. Nous proposons donc une troisième approche qui consiste à exposer les connexion internes aux composites dont les *rôles* sont partiellement remplis. C'est ce concept de connexion dont les rôles peuvent encore être remplis qu'on nomme *connexion ouverte*. Par opposition, on parlera de *connexion fermée* quand une connexion n'est pas exposée et que l'ensemble des ports qui remplissent ses rôles sont donc connus.

Lorsque deux composites exposent une connexion chacun, on pourrait les faire interagir en instanciant un composant et en le faisant participer aux deux connexions. Cette approche pose toutefois le même problème que les solutions précédemment décrites, elle introduit un composant qui n'est pas sémantiquement significatif dans l'assemblage.

Nous proposons donc une solution qui consiste à décrire les interactions entre de tels composites par la *fusion* des connexions qu'ils exposent comme représenté sur le schéma de la figure 5.6. Une opération de fusion de connexions génère une unique connexion logique dont chaque *rôle* est rempli par l'union des ensembles de ports qui remplissaient ce *rôle* dans les connexions fusionnées.

Cette définition implique qu'une fusion n'est possible qu'entre des connexions portées par un même connecteur. Il est aussi nécessaire que la connexion résultante soit valide. Dans le cas où elle n'est pas exposée à l'extérieur de l'assemblage, c'est c'est une connexion fermée et elle doit posséder au moins une mise en œuvre. Dans le cas où elle est exposée à l'extérieur de l'assemblage, c'est une connexion ouverte et il doit exister au moins un type de connexion avec laquelle elle puisse être fusionnée pour former une une connexion fermée valide.

L'ensemble de ces changement dans la description de l'interface des composants implique que les ports sont dorénavant des concepts internes aux composants qui ne sont exposés que par leur participation à des connexions.

Adaptateurs de connexions Les modifications proposées permettent aux composants de n'exposer qu'une entité par interaction logique quelle que soit la complexité de l'assemblage qu'ils contiennent. Ces modifications permettent aussi d'obtenir des connexions auxquelles

participent tous les primitifs impliqués, sans artefacts intermédiaire lors du choix de leurs mises en œuvres des connecteurs. Un troisième critère avait cependant été identifié pour permettre aux composants de posséder plusieurs mises en œuvres interchangeables. Il est nécessaire que lorsque deux connexions ouvertes sont interchangeables, c'est à dire qu'elles peuvent être fusionnées avec les même connexions, elles aient le même type.

Pour étudier si l'approche proposée répond à ce critère, il faut tout d'abord définir ce qu'est le type d'une connexion ouverte. Le connecteur qui porte la connexion fait partie de son type puisqu'il a été noté que seules des connexions portées par le même connecteur peuvent être fusionnées. De plus, il a été noté que la connexion fermée qui résulte de l'application des fusions doit posséder au moins une mise en œuvre. Comme ces mises en œuvres peuvent restreindre le nombre et le type des ports qui remplissent les *rôles* de la connexion, ces informations font aussi partie du type de la connexion. Pour résumer, le type d'une connexion ouverte est constitué du connecteur qui la porte ainsi que de l'ensemble des types de port associés à ses *rôles*. Cette définition est consistante avec un type générique pour lequel les types des ports sont des paramètres.

Cette définition pose toutefois problème, en effet le nombre et le type des ports qui participent à une connexion peuvent varier en fonction de sa mise en œuvre. Pour reprendre l'exemple du partage de mémoire, le nombre de ports qui remplissent le *rôle* de la connexion exposée par chaque composite est lié au degré de parallélisme du composant. De manière encore plus flagrante dans le cas de l'appel de méthodes, le type des ports qui remplissent chaque *rôle* est différent dans le cas d'une mise en œuvre séquentielle et d'une mise en œuvre parallèle.

Malgré ces différences, les connexions de ces exemples sont interchangeables car des générateurs existent qui supportent toutes les combinaisons possibles. Dans le cas de l'appel de méthode par exemple, il existe des mises en œuvres du connecteur pour toutes les combinaisons entre utilisateur séquentiel et parallèle et fournisseur séquentiel et parallèle.

Une question se pose cependant lors de l'ajout de nouvelles variations des interactions existantes. On pourrait par exemple imaginer l'ajout d'une version de l'interaction par appel de méthode qui implique plusieurs fournisseurs indépendants du service dans un but de répartition de charge. Dans ce cas il n'est pas suffisant de proposer un connecteur qui gère le cas d'un appelant séquentiel et d'un fournisseur qui s'appuie sur ce type de répartition de charge. Afin de gérer toutes les possibilités, il faut aussi gérer le cas où l'utilisateur est parallèle et on peut se poser la question de la représentation du cas où l'un des fournisseurs entre lesquels la charge est répartie est parallèle.

Pour gérer toute les combinaisons, on voit qu'il faut développer un nombre de générateurs proportionnel au produit du nombre de variations pour chaque rôle. Cette approche implique une quantité de travail très importante dès que le nombre de variation grandit comme on peut imaginer que ce serait le cas pour des interactions classiques comme l'appel de méthode. De plus elle implique qu'il est nécessaire de connaître toutes les variations existantes autour d'un connecteur pour en proposer une nouvelle ce qui rend impossible le développement indépendant de nouvelles variations.

Afin de résoudre ce problème, nous proposons d'introduire une nouvelle abstraction spécialement destinée à permettre à une connexion d'un type donné de se comporter comme une connexion d'un autre type, c'est à dire à rendre les connexions ouvertes polymorphes. Le concept d'*adaptateurs de connexion* est constitué d'un assemblage qui utilise la connexion effectivement présente pour exposer une connexion du type attendu.

Il s'agit en fait d'un cas particulier de mise en œuvre de connecteur qui expose une connexion ouverte. Lorsqu'une mise en œuvre de composant expose une connexion ouverte, elle doit être du type défini dans l'interface du composant ou si ce n'est pas le cas, il doit exister un adaptateur qui met en œuvre une connexion de ce type en exposant une connexion du type attendu.

Il s'agit en fait de reproduire le contournement qui avait été présenté sur le schéma de la figure 5.4. Toutefois, contrairement au cas où les composants non fonctionnels d'adaptation étaient inclus dans le composite, leur identification au sein d'un adaptateur de connexion permet de ne les utiliser que lorsque c'est nécessaire.

Pour cela, lors de la transformation les connexions sont tout d'abord fusionnées en ignorant les adaptateurs de connexions. Si aucune mise en œuvre n'est trouvée pour une connexion donnée, les adaptateurs de connexions sont utilisés pour insérer les artefacts nécessaire à sa mise en œuvre. De cette manière, les connexions continuent à relier directement les composants primitifs et des composants ne sont insérés pour l'adaptation que s'il n'existe pas de mise en œuvre optimisée. Cette approche permet d'assurer la compatibilité entre connexions à l'aide d'un nombre d'adaptateur qui n'est lié qu'à la somme du nombre de variations et non plus à son produit tout en permettant l'utilisation de mises en œuvres optimisées si elles existent.

Bundles L'utilisation de connecteurs permet de construire des abstraction d'interaction de plus haut niveau au dessus des abstractions existantes. Ces abstractions s'appuient sur la coopération d'un ensemble de ports dont le type est supporté par le modèle sous-jacent et qui remplissent les *rôles* du connecteur.

Pour supporter l'interaction par passage de message entre les membres d'un composant parallèle, la mise en œuvre proposée dans [BP07] s'appuie par exemple sur une interaction où un ensemble d'instances de composant communiquent par des appels de méthodes entre pairs. Chaque instance de composant est à la fois fournisseur et utilisateur d'un service et elle peut appeler une méthode sur l'instance de son choix dans l'ensemble.

Cette interaction pourrait être supportée par un connecteur comportant deux *rôles* : un pour les ports qui correspondent à l'utilisation du service et l'autre pour ceux qui correspondent à sa fourniture. Ce choix ne permet cependant pas aux mises en œuvres de la connexion d'utiliser l'information que chaque port qui fournit le service est logiquement relié à un port qui l'utilise. Pour cela, il est nécessaire d'introduire un concept de type de port utilisé pour regrouper d'autre ports : un *bundle*. Un *bundle* est utilisé pour regrouper des ports nommés de la même manière qu'une structure regroupe des valeurs nommées dans des langages comme le C par exemple.

Il faut cependant noter qu'avec la modification proposée précédemment, les ports ne sont plus exposés par les composants et qu'il est donc impossible de les regrouper directement au sein d'un *bundle*. Avec cette approche, ce sont des connexions ouvertes qui sont exposées par les composants. Les *bundles* sont modifiés en conséquence pour regrouper un ensemble de connexions ouvertes nommés.

Les *bundles* peuvent être utilisés de deux manières au sein des générateurs suivant qu'on les utilise comme de simples ports ou qu'on s'appuie sur leur contenu. Lorsqu'un *bundle* rempli un *rôle* d'une connexion, il peut être utilisé par le générateur comme les ports primitifs, en le faisant participer à l'une des connexions internes. La seconde option consiste à utiliser les connexions ouvertes qui constituent le *bundle* pour les fusionner avec d'autres connexions internes au générateur.

5.2.2 Modélisation des concepts

Cette section a proposé une nouvelle approche pour ajouter le concept de connecteur à des modèles de composants qui s'appuie sur les concepts de connexion ouverte, d'adaptateur de connexion et de *bundle*. La suite de la section présente les modifications à apporter au métamodèle présenté dans les chapitres 3 et 4 pour inclure ces concepts ainsi que la syntaxe textuelle utilisée pour leur description. Le modèle de composants ainsi modélisé est appelé *High Level Component Model* (HLCM).

Ports et connexions Le principal concept qui distingue HLCM des modèles de composants tels que décrits dans les chapitres précédents et le concept de connecteur. La classe `Connector` décrite sur le schéma de la figure 5.7 est introduite pour modéliser ce concept. Les connecteurs sont génériques, et la classe `Connector` hérite donc de la classe `GenericTypeDefinition` présentée au chapitre précédent. Il s'agit cependant d'un cas particulier puisque les paramètres génériques des connecteurs doivent être des ensemble de type de ports ou

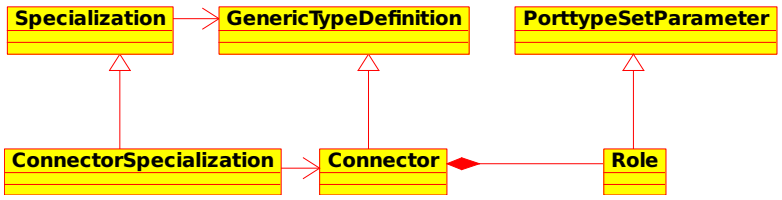

FIGURE 5.7 – Présentation de la modélisation du concept de connecteur.

```
connector UseProvide<
    role user, role provider>;
```

FIGURE 5.8 – Déclaration d'un connecteur `UseProvide` avec deux *rôles* `user` et `provider`.

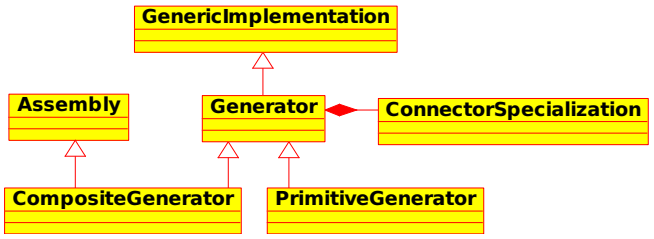

FIGURE 5.9 – Présentation de la modélisation du concept de générateur.

plus exactement des *rôles* modélisés par la classe `Role`. Un exemple de description textuelle de connecteur est présenté sur la figure 5.8.

Comme les composants, les connecteurs possèdent une ou plusieurs mises en œuvres appelées générateurs et qui sont modélisées par la classe `Generator` présenté sur le schéma de la figure 5.9. Comme pour les autres mises en œuvres de type génériques, cette classe hérite de `GenericTypeImplementation`, ce qui lui permet d'imposer des contraintes sur la valeurs des paramètres génériques du type mis en œuvre, c'est à dire sur le type des ports qui participent à la connexion.

Les générateurs peuvent être soit primitifs soit composites, dans les deux cas ils sont modélisés par une classe qui hérite de la classe `Generator`. Dans le premier cas, c'est la classe `PrimitiveGenerator` qui est utilisée, elle est destinée à être étendue lors du choix du modèle sous-jacent. Dans le second cas, c'est la classe `CompositeGenerator` qui est utilisée, elle hérite aussi de la classe `Assembly` pour décrire le fait que ces générateurs sont constitués d'un assemblage de composants.

Composants Une seconde spécificité de HLCM concerne l'interface des composants qui est constituée d'un ensemble de connexions ouvertes modélisées par la classe `OpenConnection` présentée sur le schéma de la figure 5.10. Une connexion ouverte est une instance de connecteur. Elle est typée par une spécialisation de connecteur modélisée par la classe `ConnectorSpecialization`. Les types de l'ensemble des ports qui remplisse ses *rôles* sont portés par la classe `FulfilledRole`. Puisqu'un *rôle* est un cas particulier de paramètre générique, la classe `FulfilledRole` constitue un cas particulier d'arguments générique et hérite de `PorttypeSetArgument`. Un exemple de définition textuelle d'un composant est présenté sur la figure 5.11.

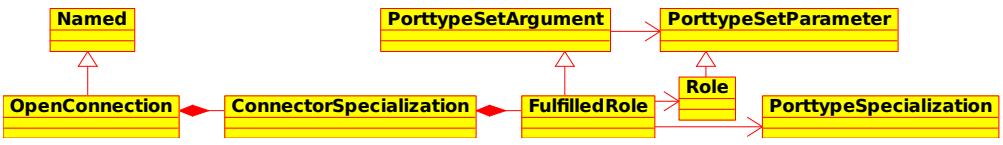

FIGURE 5.10 – Présentation de la modélisation du concept de connexion.

```
component MyHlcmComponent exposes {
  UseProvide<provider={Facet<A>}, user={}> ocA;
}
```

FIGURE 5.11 – Description textuelle d'un composant qui expose une unique connexion ouverte `ocA` porté par le connecteur `UseProvide` dont le *rôle* `provider` est rempli par un port et dont le *rôle* `user` n'est pas rempli.

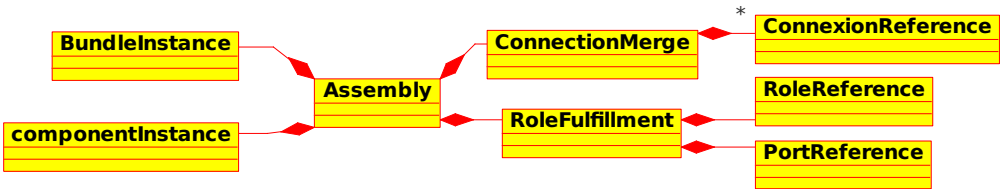

FIGURE 5.12 – Présentation de la modélisation du concept d'assemblage.

Assemblages Une autre spécificité de HLCM réside dans la manière dont les assemblages des composants utilisés notamment pour les mises en œuvres composites de composants et de connecteurs y sont décrits. La classe `Assembly` utilisée pour les modéliser est présentée sur le schéma de la figure 5.12. Un assemblage contient des instances de composant modélisées par la classe `ComponentInstance` et des instances de *bundle* modélisées par la classe `BundleInstance` qui sera présentée plus en détail plus tard.

Un assemblage contient aussi des opérations qui s'appliquent à ces instances modélisées par les classes `ConnectionMerge` et `RoleFulfillment` (à ne pas confondre avec la classe `FulfilledRole` présentée précédemment).

Les `ConnectionMerges` modélisent la fusion d'un ensemble de connexions. Les connexions fusionnées peuvent faire partie de celles exposées par les instances de composants ou de *bundle* internes à l'assemblage ou faire partie de l'interface de l'assemblage.

Les `RoleFulfillments` modélisent l'utilisation d'un port pour remplir un *rôle* de l'une des connexion de l'assemblage. Le port utilisé peut être une instance de *bundle* interne à l'assemblage ou faire partie de l'interface de l'assemblage.

Deux exemples de représentation textuelles d'assemblages sont présentés dans le cadre d'une mise en œuvre composite de composant sur la figure 5.13 et d'un générateur composite à la figure 5.14.

Bundles Le concept de *bundle* permet de regrouper un ensemble de connexions ouvertes qui participent à une seule interaction de plus haut niveau. Ils sont modélisés par la classe `BundleTypeDefinition` présentée sur le schéma de la figure 5.16. Comme un bundle est un type de port particulier, cette classe hérite de `PorttypeDefinition`. Elle contient un ensemble

```
composite MyCompositeImplementation implements MyHlcmComponent {
  AnotherComponent c1;
  AThirdComponent c2;
  merge ( c1.ocB, c2.ocC );
  merge ( this.ocA, c1.ocA );
}
```

FIGURE 5.13 – Représentation textuelle d'une mise en œuvre composite de composant `MyCompositeImplementation` qui met en œuvre le composant `MyHlcmComponent`. L'assemblage contient deux instantes de composants `c1` et `c2` et deux fusions de connexions décrites à l'aide du mot clef **merge**.

```
generator LoggingUP<UI,PI> implements
   UseProvide<provider={Facet<PI>}, user={Receptacle<UI>}>
when ( UI super PI ) {
  LoggerComponent<UI> proxy;
  proxy.clientSide.user += this.user[0];
  proxy.serverSide.provider += this.provider[0];
}
```

FIGURE 5.14 – Représentation textuelle d'un générateur composite `LoggingUP`. Il met en œuvre le connecteur `UseProvide` lorsque le *rôle* `provider` est rempli par un unique port de type `Facet<PI>` et que le *rôle* `user` est rempli par un unique port de type `Receptacle<UI>`. Il impose en plus la contrainte que l'interface `PI` hérite de `UI`. L'assemblage utilisé comme mise en œuvre contient une instance du type de composant `LoggerComponent<UI>` nommée `proxy`. Les deux ports qui participent à la connexions sont utilisées pour remplir des ports de connexions internes à l'aide du symbole **+=**.

```
bundletype CcmPeer<porttype I> {
  UseProvide<provider={Facet<I>}, user={}> pc;
  UseProvide<provider={}, user={Receptacle<I>}> uc;
}
```

FIGURE 5.15 – Représentation textuelle d'un *bundle* `CcmPeer` qui contient deux connexions ouvertes : `pc` et `uc`. Un tel *bundle* pourrait être utilisé pour décrire des connexions entre pairs qui sont à la fois fournisseur potentiel et utilisateur d'un service donné.

de connexions ouvertes qui définissent le type des connexions que contient le *bundle*. Les instances de *bundle* sont modélisées comme les autres instances de ports par la classe `PortInstance` non représentée. Un exemple de représentation textuelle d'un *bundle* est présenté sur la figure 5.15.

Adaptateurs de connexion Le dernier concept introduit dans HLCM est celui d'adaptateur de connexion qui est modélisé par la classe `ConnectionAdaptor` présentée sur le schéma de la figure 5.17. Comme un générateur composite, un adaptateur de connexion est une mise en œuvre générique qui met en œuvre une spécialisation de connecteur par un assemblage. La classe `ConnectionAdapter` hérite donc de `GenericImplementation` et de `Assembly` et contient une `ConnectorSpecialization` qui spécifie la spécialisation mise en œuvre.

À la différence d'un générateur composite cependant, un adaptateur de connexion expose une connexion ouverte modélisée par une instance de `OpenConnection` et qui représente le type qu'il permet de remplacer. Les adaptateurs de connexions constituent un type particulier puisqu'il n'existe pas de descripteur d'instance de ce type. Ils sont automatiquement utilisés si le type d'une connexion exposée par une mise en œuvre composite de composant

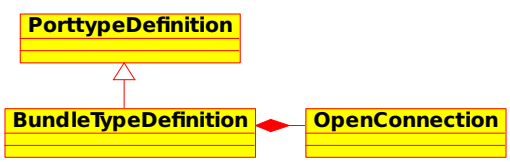

FIGURE 5.16 – Présentation de la modélisation du concept de bundle.

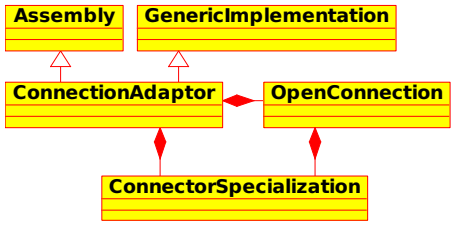

FIGURE 5.17 – Présentation de la modélisation du concept d'adaptateur de connexion.

```
adaptor PushPull supports
    UseProvide<user={Receptacle<Push>},provider={}> //< supported
    as UseProvide<user={}, provider={Facet<Pull>}> //< this
{
  BufferComponent buffer;
  merge(buffer.pushSide, supported);
  merge(this, buffer.pullSide);
}
```

FIGURE 5.18 – Description textuelle d'un adaptateur de connexion qui met en œuvre une connexion de type `UseProvide` (`supported`) dont le *rôle* `user` est rempli par un port de type `Receptacle<Push>` en exposant à la place une connexion `UseProvide` (`this`) dont le *rôle* `provider` est rempli par un port de type `Facet<Pull>`. Un tel adaptateur de connexion pourrait être utilisé pour supporter de façon transparente des interactions par flux de données où les composants utilisent des approches différentes pour le transfert des données.

ne correspond pas au type attendu.

Un exemple de description textuelle d'adaptateur de connexion est présenté sur la figure 5.18.

5.2.3 Support à l'exécution

Afin de supporter les concepts spécifiques à HLCM introduits précédemment, ceux-ci doivent être pris en compte par la transformation appliquée lors du déploiement de l'application. En fournissant une description qui s'appuie sur ces concepts à l'opération de transformation, elle doit générer une description qui est une instance du modèle sous-jacent. Il s'agit comme pour les opérations de transformations présentées dans les chapitres précédents de remplacer chaque instance d'un concept de haut niveau par une ou plusieurs instances de concepts primitifs.

Les concepts de haut niveau qui doivent être supportés sont ceux qui ont déjà été introduits dans les chapitres précédents —les mises en œuvres composites de composants et la généricité— augmentés de ceux présentés dans ce chapitre : les connecteurs et les types de ports comme entités de première classe et les adaptateurs de connexions.

Le choix d'utiliser la notion de connexion ouverte pour décrire l'interface des composants nécessite une première transformation pour identifier les instances complètes de connexion avec l'ensemble des ports qui y participent. Il est aussi nécessaire de choisir la mise en œuvre la plus appropriée pour chaque connexion. Les types non primitifs utilisées dans l'assemblage tels que les *bundles*, les générateurs composites ou les adaptateurs de connexions doivent être remplacées par leur mise en œuvre jusqu'à ce qu'il ne reste plus que des instance de types primitifs. Finalement, les types primitifs introduits dans ce chapitre —types de ports primitifs et générateurs primitifs— qui ne correspondent pas à des entités de première classe dans le modèle sous-jacent doivent être supportés.

Fusion des connexions La fusion des connexions consiste à appliquer les fusions décrites au sein des composants pour obtenir une unique connexion à partir de l'ensemble des connexions ouvertes qui la compose. Comme précédemment spécifié, chaque fusion d'un ensemble de connexions génère une connexion dont chaque *rôle* est rempli par l'union des ensembles de ports qui remplissaient ce *rôle* dans les connexions fusionnées.

Pour pouvoir appliquer les adaptateurs de connexion nécessaires dans le cas où il n'existe pas de mise en œuvre du résultat direct de la fusion, il est nécessaire de déterminer des adaptateurs qui peuvent être utilisés. Un adaptateur peut être utilisé lorsque le type effectif d'une connexion ne correspond pas au type décrit dans l'interface du composant qui l'expose. Pour pouvoir déterminer le type effectif des connexions exposées par les composants,

il est nécessaire d'effectuer l'ensemble des fusions à l'intérieur de chaque composite avant de l'exposer.

Les fusions de connexion ne doivent donc être appliquées que quand la hiérarchie complète des instances de composant est connue et commencer au bas de la hiérarchie pour remonter jusqu'à sa racine. La modélisation des connexions résultant d'une telle fusion doit comporter les informations relatives aux ports qui participent aux connexions et qui avec le connecteur déterminent le type effectif de la connexion. Cette modélisation doit aussi comporter les informations permettant de déterminer les adaptateurs de connexion qui peuvent être appliqués à la connexion.

Choix des mises en œuvres Une fois les connexions construites, il est nécessaire de choisir leurs mises en œuvres parmi celles disponibles. Pour cela, il est nécessaire de connaître leur type, c'est à dire le connecteur utilisé et les types des ports qui y participent. Des contraintes additionnelles qui concernent par exemple le placement des instance de composant qui exposent ces ports sur les ressources d'exécution peuvent aussi être exprimés par les générateurs. Ces contraintes doivent être propagées aux instances sur lesquelles elles s'appliquent.

Dans le cas où aucun générateur valide n'est trouvé, les éventuels adaptateurs dont il a été déterminé qu'ils pouvaient s'appliquer à la connexion sont utilisés et une mise en œuvre est recherchée pour la connexion résultante. Si cette dernière étape ne permet pas d'obtenir une mise en œuvre utilisable, l'assemblage est mal-formé par définition et la transformation se termine en erreur.

Dans le cas général, il est aussi possible que des contraintes incompatibles entre elles soient générées. On pourrait par exemple imaginer que le choix d'un générateur pour une connexion impose à deux instances de composants d'être localisés sur des grappes de calcul différentes alors que le générateur utilisé pour une autre connexion impose que ces mêmes instances partagent un même espace d'adressage.

La difficulté de faire des choix qui n'amènent pas à de telles contradiction varie en fonction des contraintes qui peuvent être exprimées, c'est à dire en fonction de la spécialisation utilisée. Diverses solutions peuvent être envisagées pour résoudre cette difficulté qui vont de la restriction des contraintes exprimables à un ensemble qui empêche de telles contradictions d'apparaître à un mécanisme de retour en arrière pour remettre en cause les choix effectués. Cette question est discutée dans le chapitre 6 dédié à une mise en œuvre de HLCM.

Support des types non primitifs Les types non primitifs qui peuvent apparaître dans la description d'application de haut niveau sont —en plus des mises en œuvres composites de composants— les *bundles*, les générateurs composites et les adaptateurs de connexion. Leur support est très similaire à celui des mises en œuvres composites de composant : il s'agit d'en remplacer chaque occurrence par le contenu de sa mise en œuvre. Les *bundles* sont remplacés par les connexions qui les composent et les connexions mises en œuvre par des générateurs composites ou des adaptateurs de connexion par l'assemblage correspondant.

L'application de cette transformation pour les connexions mises en œuvre par des générateurs composites ou des adaptateurs de connexion donne naissance à une nouvelle hiérarchie d'instances de composants interne à chaque connexion. La transformation doit être appliquée récursivement à chacune de ces nouvelles hiérarchies jusqu'à ce que seuls des instances de mise en œuvre primitives apparaissent dans le résultat.

Support des types primitifs L'introduction des types de ports et générateurs primitifs répond au besoins de pouvoir modéliser les ports et les connexion comme une instance d'un type, indépendamment du fait qu'ils soit nativement supportés par le modèle sous-jacent ou non. À la différence des instances de composant dont de nouveaux types primitifs peuvent être décrits par l'utilisateur, ces types ne sont pas des entités de première classe du modèle sous-jacent. Il n'est donc pas possible pour l'utilisateur de décrire de nouveaux

Algorithme 1 Algorithme de transformation d'une application décrite en HLCM vers sa version primitive.

tant que il reste des instances de composant dont la mise en œuvre n'a pas encore été choisie **faire**
  pour tout instance de composant dont la mise en œuvre n'a pas encore été choisie **faire**
    Effectuer le choix d'une mise en œuvre pour cette instance.
  fin pour
  pour tout instance de composant dont la mise en œuvre est composite **faire**
    Remplacer l'instance par l'assemblage qui la met en œuvre.
  fin pour
fin tant que
Appliquer les fusions de connexions selon une approche de bas en haut.
pour tout connexion dont la mise en œuvre n'a pas encore été choisie **faire**
  si c'est possible **alors**
    Effectuer le choix d'une mise en œuvre pour cette connexion.
  sinon
    Appliquer les adaptateurs de connexion.
    Effectuer le choix d'une mise en œuvre pour la connexion résultante.
  fin si
fin pour
pour tout connexion mise en œuvre par un générateur composite **faire**
  Remplacer la connexion par l'assemblage qui la met en œuvre.
  Appliquer récursivement l'algorithme à cet assemblage.
fin pour
pour tout connexion ou port primitif **faire**
  Replacer cette représentation par la classe dédiée.
fin pour

ports ou générateurs primitifs. Chaque spécialisation de HLCM inclue à la place un nombre fini d'instances de ces classes qui modélise les types supportés par le modèle sous-jacent.

Pour obtenir exactement le même résultat que ce qui aurait été obtenu sans l'introduction de ces concepts comme entité de première classe, la représentation des instances de mise en œuvre primitive doit être transformée. Cette transformation est spécifique à chaque spécialisation et consiste à transformer les instances qui référencent un type primitif en des instances de classes dédiées. Par exemple pour le cas d'une connexion CCM, il s'agit de transformer chaque instance de la classe `Connexion` qui référence comme type l'instance `UseProvide` de la classe `PrimitiveGenerator` en une instance de la classe `UseProvideConnexion`.

Algorithme complet Finalement, l'algorithme 1 décrit les étapes nécessaires à la transformation complète. Il s'agit d'un algorithme qui reste relativement simple. La seule difficulté réelle consiste à gérer les interactions possibles entre les contraintes des générateurs et il s'agit d'un point spécifique à chaque spécialisation.

5.2.4 Conclusion

Cette section a proposé une approche pour introduire le concept de connecteur comme entité de première classe au sein de modèles de composants hiérarchiques. Cette approche s'appuie sur les concepts de connexion ouverte pour décrire l'interface des composants, de *bundles* et de générateurs composites pour construire des connexions de haut niveau d'abstraction au dessus de celles existantes et d'adaptateur de connexion pour supporter le polymorphisme des connexions ouvertes. Cette approche a été appliquée pour donner

```
//IDL3 OMG annoté pour HLCM/CCM
//@implements MyHlcmComponent
component MyCcmImplementation {
  //@fulfills ocA.provider
  provides A a_port;
}
```

FIGURE 5.19 – Exemple en IDL OMG étendu d'une mise en œuvre CCM (primitive) du composant `MyHlcmComponent`. Le port CCM `a_port` rempli le *rôle* `provider` de la connexion ouverte `ocA`.

HLCM, un modèle de composants abstrait dont la modélisation et le support par une opération de transformation lors du déploiement ont été décris.

La section suivante propose une première validation des choix qui ont été effectués en utilisant HLCM pour mettre en œuvre les exemples synthétiques d'applications présentés en 5.1.2.

5.3 Validation du modèle

Afin de valider les choix effectués pour HLCM, cette section s'appuie sur une spécialisation qui utilise CCM comme modèle sous-jacent. Cette spécialisation est utilisée pour mettre en œuvre les deux variation de l'exemple présenté à la section 5.1.2 et pour illustrer leur comportement en déroulant l'application de la transformation sur ces deux exemples.

5.3.1 Spécialisation de HLCM pour CCM : HLCM/CCM

Mises en œuvres de composants Les mises en œuvres primitives de composants de HLCM/CCM sont des composants CCM dont les ports sont utilisés pour remplir les *rôles* des connexions exposées par le composant HLCM. Un langage qui étend l'IDL OMG avec des annotations dédiées est utilisé pour définir les composants HLCM/CCM comme présenté sur la figure 5.19

Types de ports et générateurs Les types de ports et les générateurs primitifs de HLCM/CCM sont ceux de CCM. Ils existent en nombre finis.

Les interactions par appel de méthodes à distance sont supportées avec deux types de ports : les `facets` et les `receptacles`, chacun paramétré par une interface objet CORBA. La connexion de ces ports est mise en œuvre par le générateur `CcmUseProvide` qui met en œuvre le connecteur `UseProvide` dont le *rôle* `user` est rempli par un `receptacle` et le *rôle* `provider` par une `facet` dans le cas où l'interface qui paramètre la `facet` hérite de celle qui paramètre le `receptacle`.

Les interactions par passage d'événements s'appuient sur les types de ports `publishers`, `emitters` et `sinks`, chacun paramétré par un type d'événement CORBA. La connexion de ces ports est mise en œuvre par deux générateurs primitifs. Le générateur `PublishedEventChannel` met en œuvre le connecteur `EventChannel` dont le *rôle* `emmitter` est rempli par exactement un `publisher` et le *rôle* `receiver` par des `sinks` dont le type d'événement est compatible. De manière similaire, le générateur `EmmittedEventChannel` met en œuvre le connecteur `EventChannel` le *rôle* `receiver` est rempli par un unique `sink` et le *rôle* `emmitter` par des `emitters` dont le type d'événement est compatible.

Ces générateurs correspondent au connexions directement supportées entre les ports de composants CCM.

```
connector SharedMem<role access>;
```

FIGURE 5.20 – Déclaration du connecteur SharedMem comportant un unique *rôle* : access.

```
interface DataAccess
{
    // création d'une zone mémoire avec identifiant
    void share(in unsigned long size, out datapointer data, in
 string id);

    // récupération des informations sur une zone mémoire associée à un identifiant
    void access(in string id, out datapointer data, out unsigned
 long size);

    // découplage d'une zone mémoire et de son identifiant
    void detach(in string id);

    // manipulation des verrous associés aux zones mémoire
    void write_lock(in string id);
    void read_lock(in string id);
    void release(in string id);
};
```

FIGURE 5.21 – Déclaration en IDL OMG de l'interface DataAccess.

5.3.2 Mise en œuvre de l'exemple avec interaction par partage de mémoire avec HLCM/CCM

Description L'interaction par partage de mémoire s'appuie sur le connecteur dédié SharedMem présenté sur la figure 5.20. Ce connecteur comporte un unique *rôle* accessor destiné à regrouper tous les composants qui participent à l'interaction.

Pour accéder à la mémoire partagée, il est nécessaire d'utiliser une interface permettant de manipuler son contenu. Pour représenter cet état de fait, la participation à la connexion se fait par un receptacle CCM paramétré par une interface objet spécifique. Cette interface présentée sur la figure 5.21 et nommée DataAccess possède la spécificité de permettre d'accéder à un pointeur sur la mémoire porté par le type DataPointer qui correspond à un void* en C.

Ce choix implique qu'il est nécessaire que les deux instances de composant qui interagissent par cette interface soit localisés dans le même espace d'adressage, c'est à dire le même processus. Afin de permettre la spécification de contraintes de ce type, HLCM/CCM a été étendu avec deux nouveaux types de port primitifs : LocalReceptacle et LocalFacet. Ces deux types se comportent exactement comme les types Receptacle et Facet à la différence près que le générateur qui met en œuvre leurs connexions (LocalCcmUseProvide) impose aux composants qui interagissent au travers de ces ports d'être situés dans le même processus. En intégrant la contrainte de collocation à la connexion exposée par les composants, cette approche évite de devoir ajouter des contraintes au moment de la connexion et de risquer de l'oublier.

Un composant peut alors utiliser la mémoire partagée de manière sûre au travers d'un port de type LocalReceptacle<DataAccess> utilisé pour remplir le *rôle* accessor d'une connexion SharedMem comme présenté sur la figure 5.22.

Ces choix pour la description de l'interaction par partage de mémoire permettent de décrire une mise en œuvre composite de composant comme CompositeAccessorImpl présentée sur la figure 5.23 qui spécifie que les instances de composant qui la composent participent à une même connexion tout en permettant à d'autres instances externes d'y participer.

```
component MemoryAccessor exposes {
  SharedMem<access={LocalReceptacle<DataAccess>}> memory;
}
```

FIGURE 5.22 – Déclaration d'un composant `MemoryAccessor` qui expose une connexion ouverte `memory` dont le *rôle* `access` est rempli par un port de type `LocalReceptacle<DataAccess>`.

```
composite CompositeAccessorImpl implements MultiAccessor {
  MemoryAccessor c1;
  MemoryAccessor c2;
  merge(this.memory, c1.memory, c2.memory);
}
```

FIGURE 5.23 – La mise en œuvre composite `CompositeAccessorImpl` fusionne les connexions `c1.memory` et `c2.memory` avec la connexion exposée `this.memory`.

5.3.3 Mise en œuvre de l'exemple avec interaction par appel de méthode avec HLCM/CCM

Description Contrairement à l'interaction par partage de mémoire, il n'est pas nécessaire de décrire de nouveau connecteur pour gérer l'interaction par appel de méthode : il s'agit d'un des connecteurs nativement supportés par HLCM/CCM. Il est cependant nécessaire de supporter de nouvelles mises en œuvres de ce connecteur quand l'un des deux ou les deux *rôles* sont remplis non plus par une unique instance de composant primitif mais par un ensemble d'instances qui coopèrent au sein d'une mise en œuvre parallèle de composant.

Pour représenter le fait que plusieurs instances de composant coopèrent pour fournir ou utiliser le service, deux options sont possibles :

1. laisser le *rôle* correspondant être rempli plusieurs fois, une fois par participant ; ou
2. regrouper les ports au sein d'un *bundle* et utiliser ce *bundle* pour remplir le *rôle* une seule fois.

Cette seconde approche a l'avantage de permettre d'associer une sémantique portée par le *bundle* au regroupement des ports. Ce choix permet des regroupements à plusieurs niveaux, comme par exemple en regroupant dans le but d'un équilibrage de charge plusieurs fournisseurs du service eu mêmes parallèles et donc constitués d'un ensemble d'instances qui coopèrent.

Deux *bundles* `ParallelFacet` et `ParallelReceptacle` sont donc introduits pour représenter une coopération pour fournir le service ou pour l'utiliser respectivement. Ces *bundles* sont paramétrés par une interface objet à la manière de leurs pendants séquentiels et par un entier qui représente le nombre de participants à l'interaction. L'exemple de la spécification du *bundle* `ParallelFacet` est présenté sur la figure 5.24.

Quatre combinaisons valides existent pour remplir les *rôles* `user` et `provider` du connecteur `UseProvide` suivant que le *rôle* `user` est rempli par un port de type `Receptacle` ou `ParallelReceptacle` et que le *rôle* `provide` est rempli par un port de type `Facet` ou `ParallelFacet`. Parmi ces combinaisons, une est nativement gérée par HLCM/CCM : le cas où les deux participants sont séquentiels.

```
bundletype ParallelFacet<Integer N, interface I> {
  each(i:[1..N]){ UseProvide<provider={Facet<I>}> part[i]; }
}
```

FIGURE 5.24 – Définition du *bundle* `ParallelFacet` qui contient `N` connexions `UseProvide` appelées `part` et dont le *rôle* `provider` est rempli par un port de type `Facet<I>`.

```
adaptor Scatter<Integer N> supports
    UseProvide<user={ParallelReceptacle<N, MatrixPart>}>
    as UseProvide<user={Receptacle<Matrix>}> {
  Distributor<N> dist;
  each(i:[1..N]){ merge(dist.in[i], supported.user.part[i]); }
  merge(this, dist.out);
}
```

FIGURE 5.25 – Définition de l'adaptateur de connexion `Scatter` qui supporte une connexion `UseProvide` dont le *rôle* `user` est rempli par un port de type `ParallelReceptacle<N, MatrixPart>` comme si il s'agissait d'une connexion dans laquelle le *rôle* `user` est rempli par un port de type `Receptacle<Matrix>`.

Afin de gérer les trois cas supplémentaires, on pourrait envisager de décrire trois nouveaux générateurs. Comme cela a été montré plus tôt, cette approche ne permettrait pas d'assurer que les composants soient compatibles si de nouvelles variations autour de l'interaction par appel de méthodes étaient introduites. Il ne serait alors pas envisageable de décrire la version parallèle et la version séquentielle comme deux mises en œuvres d'un même composant, empêchant ainsi un choix automatique entre ces version.

Il s'agit donc d'une situation qui se prête particulièrement bien à l'utilisation d'adaptateurs de connexion. L'adaptateur `Scatter` qui permet d'exposer des connexions dont le *rôle* `user` est rempli par un port de type `ParallelReceptacle` comme si ce *rôle* était rempli par un port de type `Receptacle` est présenté sur la figure 5.25. Un connecteur `Gather` non représenté permet symétriquement d'exposer des connexions dont le *rôle* `provider` est rempli par un port de type `ParallelFacet` comme si ce *rôle* était rempli part un port de type `Facet`.

L'ajout de ces deux adaptateurs de connexion n'empêche pas de fournir des mises en œuvres optimisées pour certains cas particuliers. Dans ce cas précis, il semble par exemple intéressant de proposer un générateur qui gère le cas où à la fois l'utilisateur et le fournisseur du service sont parallèle sans introduire de goulot d'étranglement.

5.3.4 Analyse

Ces deux exemples montrent qu'il est possible de décrire les exemples présentés à la section 5.1.2 en utilisant des mises en œuvres interchangeables à la fois pour les codes qui constituent l'application et pour leurs interactions. L'approche adoptée dans HLCM se distingue ainsi de ce qui avait été observé avec l'approche utilisée dans les modèles de composants existant.

Cette possibilité est due au fait que HLCM assure trois propriétés importantes :

1. une mise en œuvre composite peut exposer un unique point d'interaction qui implique plusieurs instances de composant internes à la mise en œuvre ;
2. plusieurs mises en œuvres de composant peuvent respecter une même interface tant que les points d'interactions qu'elles exposent sont compatibles et même si ils ne possèdent pas exactement le même type ; et
3. ces deux point ne nécessitent pas l'insertion d'artefacts sans sémantique dans l'assemblage.

Ces propriétés permettent de découpler la hiérarchie des composants des interactions qui peuvent alors logiquement traverser les barrières de cette hiérarchie.

L'approche adoptée qui s'appuie sur l'utilisation d'un modèle existant (CCM) comme modèle sous jacent impose cependant des limitations. Il s'agit en effet d'un modèle qui propose déjà un niveau d'abstraction relativement élevé qui masque la distribution entre les composants en imposant l'utilisation d'appels de méthodes CORBA entre eux. Certaines interactions nécessitent l'utilisation de mécanismes de plus bas niveau comme le partage de mémoire qui s'appuie sur des transferts d'adresses mémoire. Cette possibilité n'existe pas

dans CCM et a donc nécessité l'utilisation d'un contournement qui passe par une modification du modèle sous-jacent. Ce point sera discuté plus en détail dans le chapitre 7.

5.4 Conclusion

Ce chapitre a étudié la possibilité d'intégrer le concept de connecteurs de première classe à un modèle de composants hiérarchique et générique. L'analyse d'exemples synthétiques d'applications a permis d'identifier un problème inhérente à l'approche naïve classiquement utilisée en présence de la hiérarchie. Dans certaines situations cette approche impose aux utilisateurs de faire un choix entre des mise en œuvre efficaces des connecteurs et la possibilité de remplacer les mises en œuvres de composants.

Une nouvelle approche a été proposée qui s'appuie sur trois nouveaux concepts : les connexions ouvertes, les adaptateurs de connexions et les *bundles*. Les connexions ouvertes sont utilisées pour décrire l'interface des composants, les adaptateurs de connexions supportent le polymorphisme et les *bundles* permettent de construire des abstractions de plus haut niveau au dessus de celles existantes. Avec la généricité telle que présentée au chapitre 4, ces trois concepts forment la base du modèle de composants abstrait HLCM.

Un patron de conception permettant de modéliser ces concepts a été décrit ainsi qu'une transformation de modèles permettant de supporter leur exécution. Cette approche a été appliquée à CCM pour donner HLCM/CCM qui a été utilisé pour son évaluation. Pour cela, les exemples synthétiques d'applications de la section 5.1 ont été décrits à l'aide de HLCM/CCM.

Il ressort de cette évaluation que l'approche choisie permet de décrire des interactions entre composants à l'aide de connecteurs. Une spécificité de cette approche est qu'elle permet aux connexion de relier de manière directe les composants primitifs sans empêcher l'existence de plusieurs mises en œuvres interchangeables pour les composants et les connecteurs. La possibilité de connexion directe semble offrir la possibilité de proposer des mises en œuvres performantes des connexion, la question des performances exactes pouvant être obtenues sera étudiée plus en détail dans le chapitre 7. Ces caractéristiques correspondent aux objectifs qui ont été définis au chapitre 3 pour la conception d'un modèle de composants extensible par des fonctionnalités dédiées au calcul à haute performance.

CHAPITRE 6

$HLCM_i$: une plate-forme de mise en œuvre de HLCM

Les chapitres précédents ont présenté les fonctionnalités qui forment la base de HLCM. Il s'agit du support de multiples mises en œuvres pour les composants, de la hiérarchie, de la généricité et des connecteurs comme entités de première classe. Une approche a été proposée pour supporter ces fonctionnalités qui s'appuie sur une transformation appliquée lors du déploiement de l'application. Cette transformation prend en entrée une description de l'application dans un modèle indépendant de la plateforme d'exécution et génère en sortie une description de cette même application selon un modèle spécifique à la plateforme.

Ce chapitre s'intéresse à la mise en œuvre de HLCM au sein d'un prototype nommé $HLCM_i$ (HLCM *implementation*) pour en montrer la faisabilité. Dans la mesure où le cœur de $HLCM_i$ est basé sur approche à base de modèle, un environnement dédié à ce type d'approches est utilisé : le projet de modélisation d'Eclipse. Il faut toutefois noter que HLCM est un modèle de composants abstrait et que ce sont ses spécialisations qui sont des modèles de composants à part entière. Afin de faciliter le partage de code entre les mises en œuvres de ces diverses spécialisations, $HLCM_i$ est développé sous la forme d'un ensemble de composants pouvant être utilisés par ces mises en œuvre.

Les sections 6.1 et 6.2 présentent les deux choix technologiques effectués au sein de $HLCM_i$ qui concernent l'environnement de modélisation et le modèle de composants respectivement. La section 6.3 présente l'architecture globale des mises en œuvres basées sur $HLCM_i$ et les composants qu'elles partagent. La section 6.4 présente l'utilisation de ces outils pour construire $HLCM_i$/CCM, une mise en œuvre de HLCM/CCM, la spécialisation de HLCM présentée au chapitre 5. Finalement, la section 6.5 conclue le chapitre.

6.1 Environnement de modélisation

Comme cela a été présenté au chapitre 3, HLCM s'appuie sur les concepts typiques d'une approche dirigée par les modèles qui sont explicités dans cette section. Les syntaxes abstraites des programmes décrits à l'aide de spécialisations de HLCM sont définies par des métamodèles. Une syntaxe concrète textuelle est spécifiée pour décrire la partie de ces programmes qui est indépendante de la spécialisation. Chaque spécialisation offre en plus une approche pour décrire les instances des parties du métamodèle spécifiques à la spécialisation. Un second métamodèle est utilisé pour la description de l'application adaptée à une plateforme d'exécution donnée. Lors du déploiement, une transformation de modèle est appliquée qui transforme une instance du premier métamodèle en une instance du second.

Afin de simplifier l'utilisation de cette approche, il semble avantageux de s'appuyer sur un environnement de modélisation existant. Comme pour ses autres standards, l'OMG ne propose pas de mise en œuvre officielle du MDA, elles sont proposées de manières indépendantes par d'autres acteurs. Il existe plusieurs outils qui s'inspirent de cette approche dans divers domaines.

Aujourd'hui, l'essentiel des efforts dans ce sens se concentrent autour du projet de modélisation d'Eclipse (*Eclipse Modeling Project*). Les rares outils qui ne font pas partie de ce projet se concentrent généralement sur un domaine spécifique ou constituent des prototypes pour l'évaluation d'approches différentes développés dans le cadre de projets de recherche. C'est donc la solution proposée au sein d'Eclipse qui a été choisie pour développer HLCM$_i$.

Le projet de modélisation d'Eclipse propose un environnement de modélisation qui s'appuie sur les standards OMG en s'en éloignant parfois un peu pour adopter une approche plus pragmatique. Un exemple de ce pragmatisme consiste en un lien fort avec le langage Java au lieu de l'approche plus indépendante du langage pronée par l'OMG. Les outils proposés sont aussi souvent fortement dépendants de services offerts par la plateforme Eclipse. Il faut toutefois noter qu'un effort a été fait pour restreindre cette dépendance à l'étape de développement et pour permettre l'utilisation du résultat sans lancer Eclipse et n'en utilisant que les services les plus importants distribués sous la forme de bibliothèques.

Ce projet est découpé en plusieurs sous-projets avec des objectifs distincts comme par exemple :

- le *framework* de modélisation d'Eclipse (*Eclipse Modeling Framework* – EMF) pour le développement de syntaxes abstraites, c'est à dire la modélisation proprement dite ;
- le *framework* de modélisation textuelle (*Textual Modeling Framework* – TMF) pour le développement de syntaxes concrètes textuelles ;
- le *framework* de modélisation graphique (*Graphical Modeling Project* – GMP) pour le développement de syntaxes concrètes graphiques ;
- un projet pour les transformations de modèle à modèle (*Model to Model* – M2M) ;
- un projet pour les transformations de modèle vers une représentation textuelle (*Model to Text* – M2T) ; et
- un projet pour l'ajout de contraintes sur les modèles avec le langage de contraintes sur les objets (*Object Constraint Language* – OCL).

La suite de cette section présente plus en détail les divers sous-projets utilisés au sein de HLCM$_i$.

6.1.1 Syntaxe abstraite

La partie concernant la description de la syntaxe abstraite d'un langage, c'est à dire la description de son métamodèle est gérée par le sous-projet EMF. Le cœur de ce sous-projet est constitué du méta-métamodèle Ecore (EMF *Core*) qui prend la place de MOF du standard OMG pour la modélisation de métamodèles. Contrairement à MOF, Ecore est restreint à un ensemble minimal de concepts puisqu'il ne comporte que 20 classes et il s'appuie fortement sur le langage Java dont les types forment les types primitifs d'Ecore. Il

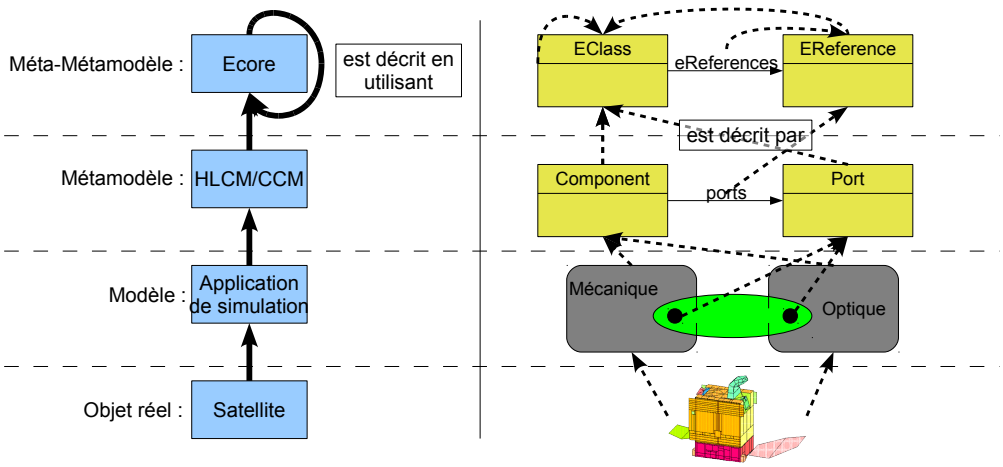

FIGURE 6.1 – Les quatre niveau de modélisation dans le cadre du projet de modélisation d'ECLIPSE.

```
class ComponenttypeArgument extends modeling.GenericArgument {
  attr String name;
  ref ComponenttypeParameter parameter;
  val ComponenttypeSpecification value;
}
```

FIGURE 6.2 – Exemple de description d'une classe ECORE en EMFATIC. La classe spécifie un attribut `name`, une référence `parameter` et une relation de contenance `value`.

faut toutefois noter que dans ses dernières révisions du standard, l'OMG a extrait un sous-ensemble de MOF qui correspond globalement à l'étendue de ECORE : EMOF (*Essential* MOF).

Comme pour MOF, ECORE est un modèle qui se situe en haut de la pile de modélisation et qui est donc utilisé pour se décrire lui même, comme le présente le schéma de la figure 6.1. Il définit donc une syntaxe abstraite et plusieurs syntaxes concrètes peuvent être utilisées pour décrire ses instances. Comme pour toutes les instances de modèle, les instances de ECORE peuvent être stockées en utilisant le dialecte de XML défini dans ce but par un standard de l'OMG : XMI (XML *Model Interchange*). Des éditeurs graphiques intégrés à ECLIPSE permettent de créer et de manipuler de manière générique les instances de modèles sous la forme d'un arbre sous-tendu par les relations de contenance du modèle.

Malgré ces éditeurs, la manipulation d'instances de modèles à l'aide d'outils non spécifique au modèle reste relativement pénible. Un projet récent propose une syntaxe proche des langages objets comme C++ ou JAVA avec lesquels ECORE possède de nombreuses similitudes pour en décrire des instances, il s'agit d'EMFATIC. Un outil permet de transformer une instance décrite en XMI vers un fichier utilisant cette syntaxe et inversement. Un exemple de classe ECORE décrite à l'aide de cette syntaxe est présentée sur la figure 6.2 et son équivalent en XMI est présenté sur la figure 6.3.

Ces deux représentations permettent de décrire un métamodèle comme celui de HLCM à l'aide du méta-métamodèle ECORE. L'intérêt d'un métamodèle réside cependant dans la possibilité de l'utiliser pour en décrire des instances, c'est à dire des applications qui sont des modèles d'objets réels. Pour cela, une troisième représentation des instances de ECORE existe sous la forme de classes JAVA annotées. Cette représentation permet de manipuler les instances du métamodèle sous la forme d'ensemble d'objets JAVA. À nouveau, un outil permet de transformer un modèle instance de ECORE décrit en XMI sous cette forme et inversement.

```
<eClassifiers xsi:type="ecore:EClass" name="ComponenttypeArgument"
    eSuperTypes="Modeling.ecore#//GenericArgument">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="parameter"
      eType="#//ComponenttypeParameter"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="value"
      eType="#//ComponenttypeSpecification" containment="true"/>
</eClassifiers>
```

FIGURE 6.3 – Exemple de description d'une classe ECORE en XMI. La classe spécifie un attribut `name`, une référence `parameter` et une relation de contenance `value`.

Ce code comporte une interface JAVA pour chaque classe du métamodèle qui comporte des accesseurs pour chaque attribut de la classe. Des classes JAVA qui mettent en œuvre ces interfaces sont aussi générées. Le patron de conception *fabrique* est utilisé pour gérer l'instanciation des classes associées à chaque interface.

Ces classes mettent aussi en œuvre une interface qui permet un accès réflectif au modèle. Elle permettent notamment d'accéder à la valeur d'un attribut en passant son nom a une méthode fixe plutôt qu'en utilisant une méthode spécifique dont le nom correspond à celui de l'attribut. Ces classes permettent aussi d'accéder aux objets JAVA du niveau de modélisation supérieur. Il est par exemple possible d'obtenir l'objet qui représente la classe ECORE correspondant à la classe JAVA manipulée et ainsi d'obtenir la liste de ses attributs par exemple.

Cet aspect réflectif permet de créer des outils qui fonctionnent quel que soit le modèle dont l'instance est manipulée. C'est par exemple le cas du code qui met en œuvre le stockage d'instances de modèle dans des fichiers XMI ou leur édition de manière graphique dans l'éditeur ECLIPSE. Ces deux outils ont déjà été mentionnés pour le cas d'ECORE qui n'est qu'un cas particulier d'instance d'ECORE. Un troisième outil de ce type a été mentionné dans le cas de GENERICSCA au chapitre 4, il permet de stocker les instances du modèle dans des fichiers XML qui respectent un schéma qui correspond globalement au métamodèle et dont les particularités sont décrites à l'aide d'annotations.

En plus de ce cœur que constitue ECORE, le projet EMF propose d'autres outils qui peuvent globalement être classés dans deux catégories. Une première catégorie regroupe les outils qui s'attachent à permettre le stockage des modèles dans divers types de bases de donnée. La seconde catégorie regroupe ceux qui sont destinés à faciliter l'accès à ces modèles soit en offrant des interfaces accessibles à distance soit en permettant l'expression de requêtes complexes.

6.1.2 Syntaxe concrète textuelle

Deux sous-projets du projet de modélisation d'éclipse proposent des outils utiles pour supporter des syntaxes concrètes textuelles de modèles exprimés en ECORE. Le sous-projet TMF regroupe les outils qui permettent de générer une instance du modèle sous la forme d'instances des classes JAVA qui lui correspondent à partir de fichiers texte. Le sous projet M2T regroupe les outils qui permettent de générer du texte à partir d'une instance de modèle sous la forme d'objets JAVA, il peut éventuellement s'agir d'une représentation textuelle du modèle.

Dans notre cas, nous nous intéressons plus particulièrement à la première de ces deux directions, c'est à dire à parcourir des fichiers texte pour générer une instance d'un métamodèle. Un outil au sein du projet TMF a pour ambition de permettre le parcours de fichiers dont la grammaire a été mise en relation avec un métamodèle pour générer une instance de ce métamodèle, il s'agit de XTEXT.

```
ComponentDescriptor:
  type=ComponenttypeSpecification name=ID ('[' indices+=DatavalueSpecification
      (',' indices+=DatavalueSpecification)* ']')? ';'
;
```

FIGURE 6.4 – Exemple d'une règle de la grammaire de HLCM en XTEXT. l'attribut `type` de la classe `ComponentDescriptor` est construit par un appel à la règle ComponenttypeSpecification.

Cet outil utilise une grammaire du langage exprimée sous une forme proche de celle de Backus-Naur étendue augmentée d'informations sur la mise en correspondance des règles de la grammaire avec le modèle. Chaque règle de la grammaire y correspond à une classe du modèle et les appels à ces règles instancient donc des objets qui sont des instances de la classe correspondant à la règle appelée. Les objets construits par les appels à d'autres règles dans la partie droite d'une règle données sont affectés à des attributs de la classe à l'aide de l'opérateur = dans le cas d'une multiplicité simple ou += sinon. Un exemple d'une règle de la grammaire de HLCM est présenté sur la figure 6.4.

6.1.3 Transformation de modèle à modèle

Le standard proposé par l'OMG pour la transformation de modèles à modèles est le langage de requête/vue/transformation (*Query/View/Transform* – QVT). Il existe deux versions de ce standard : une version déclarative (QVT *Relational* – QVTR) et une version procédurale (QVT *Operational* – QVTO). Des efforts pour la mise en œuvre de ce standard au sein du projet M2M existent mais ils ne proposent pas encore de version réellement utilisable.

Un autre standard similaire est le langage de transformation d'ATLAS (ATLAS *Transformation Language*) développé au sein de l'équipe ATLANMOD du projet INRIA ATLAS. Il a été conçu pour répondre à l'appel à propositions de l'OMG lors de la standardisation de QVT mais n'a pas été retenu tel quel. Il est constitué d'un métamodèle destiné à modéliser des transformations et d'une syntaxe associée. Le langage comporte des éléments déclaratifs et d'autres impératifs. Le support de ce langage est maintenant développé au sein du projet ECLIPSE et fait partie intégrante du sous-projet M2M.

Il faut noter que si ce projet est aujourd'hui dans un état utilisable, il s'agit d'une avancée assez récente et que ce n'était pas le cas lors des premiers développements autour de $HLCM_i$. Au vu de ces limitations, aucune de ces deux approches n'a été utilisée pour le développement des transformations de modèle à modèle dans $HLCM_i$. Le choix a été fait de manipuler directement les classes générées en JAVA.

6.2 Modèle de composants natif

Un second choix technologique concerne le modèle de composants avec lequel mettre en œuvre $HLCM_i$. Puisque nous cherchons à promouvoir HLCM, il semble aussi intéressant d'utiliser ce modèle et c'est donc la spécialisation, c'est à dire le modèle sous-jacent qu'il faut choisir. Ce choix doit se faire en fonction des besoins pour la mise en œuvre de $HLCM_i$.

6.2.1 Analyse des besoins

Dans la mesure où le choix a été fait d'utiliser le projet de modélisation d'ECLIPSE qui s'appuie sur le langage JAVA, il est important de pouvoir décrire les composants primitifs dans ce langage. Il devrait aussi être simple d'encapsuler du code existant dans un composant de ce type.

Puisque HLCM permet de construire les interactions complexes au dessus de celles du modèle sous-jacent, il n'est pas nécessaire que le modèle sous-jacent offre d'interactions de

haut niveau mais celles qu'il offre doivent être suffisantes pour construire les interactions requises. Il semble par contre important de pouvoir utiliser la généricité jusqu'au modèle primitif comme cela a été présenté dans le chapitre 4, c'est à dire que le modèle sous-jacent doit permettre la création de composants génériques.

Un autre point est que dans l'optique de la mise en œuvre de compilateurs, une gestion transparente de la distribution ne semble pas nécessaire. Dans la mesure où HLCM est pour l'instant un modèle statique, il ne semble pas non plus important de supporter d'éventuelles évolution de l'assemblage au cours de son exécution.

Finalement, puisque ce modèle doit être utilisé pour mettre en œuvre HLCM, il est nécessaire de résoudre le problème de l'initialisation. C'est à dire qu'il doit être possible de décrire des assemblages du modèle natif directement exécutables qui ne font pas intervenir l'opération de transformation de HLCM.

Parmi les modèles de composants qui permettent la description de composants en Java, la majorité comme les EJB ou SCA introduisent un niveau de complexité relativement élevé pour supporter la distribution et d'autres interactions de haut niveau. Ils nécessitent aussi généralement l'utilisation d'un moteur d'exécution qui utilise son propre chargeur de classes et dont la compatibilité avec les outils du projet de modélisation d'Eclipse peut poser problème.

Un modèle qui peut sembler intéressant est OSGi (*Open Services Gateway initiative*) utilisé notamment au sein d'Eclipse. Il faut toutefois noter que dans ce modèle, les composants sont généralement limités à une unique instance qui n'est pas créée selon la description d'un assemblage mais automatiquement quand l'un des services qu'elle fournit est requis. Il ne s'agit pas d'un modèle de composants au sens où nous l'entendons mais plutôt d'un modèle de modules indépendants pouvant être chargés par une application (*plugins*).

Nous avons donc fait le choix de développer notre propre modèle de composants Java minimaliste. Ce modèle appelé LLCM$_j$ (*Low Level Component Model* – Java) a été être spécifiquement conçu pour servir de modèle sous-jacent à HLCM.

6.2.2 Présentation de LLCM$_j$

Une instance de composant primitif est caractérisée par trois aspects principaux :

1. son type qui défini son comportement et ses interactions possible ;
2. ses éventuels paramètres génériques et leurs valeur ;
3. les interactions auxquelles l'instance participe.

Ces trois aspects correspondent à trois étapes du cycle de vie d'un composant : sa version sur l'étagère, sa configuration au sein d'un assemblage, son instanciation. Dans la mesure où LLCM$_j$ est un modèle statique, il ne faut rajouter qu'une dernière étape pour constituer le cycle de vie complet des composants LLCM$_j$: l'exécution.

En ce qui concerne le type, l'approche la plus simple consiste à s'appuyer sur les types de Java, c'est à dire les classes. Un composant LLCM$_j$ est donc constitué d'un objet Java marqué par l'annotation `@LlcmjComponent` pour le distinguer des objets qui ne constituent pas un composant.

Paramètres génériques Les paramètres génériques sont caractérisés par le fait qu'il s'agit de valeurs qui peuvent être utilisées pour configurer les composants qui sont connues au moment de son instanciation. Il faut distinguer trois catégories de paramètres génériques : ceux qui représentent un type de HLCM qui ne correspond pas à un type du modèle primitif, ceux qui représentent un type du modèle primitif et ceux qui représentent une valeur de donnée.

Dans la première catégorie, il s'agit de paramétrer la classe qui représente le composant par un type spécifique à HLCM (composant, connecteur ou type de port) qui ne correspond à rien dans le modèle primitif. Dans la mesure où ces types et leurs instances ne peuvent pas

être manipulés au niveau du modèle primitif, les utiliser comme paramètres d'un composant primitif n'a pas de signification et ils ne sont donc pas supportés par LLCM$_j$.

Dans la seconde catégorie, il s'agit de paramétrer la classe qui représente le composant par un type de donnée du modèle primitif, c'est à dire une classe Java. Deux approches peuvent être adoptées pour utiliser une classe pour en paramétrer une autre en Java suivant l'utilisation qui en est faite. Une classe peut être spécifiée soit en s'appuyant sur les paramètres génériques de Java soit à l'aide du mécanisme d'introspection en passant un objet de la classe `Class<C>`.

La première approche qui s'appuie sur les paramètres génériques de Java permet une prise en compte à la compilation. Il est par exemple possible d'utiliser un tel paramètre pour spécifier le type d'instances internes à la mise en œuvre. Il faut cependant noter que comme cela a déjà été étudié dans le chapitre 4, l'approche par effacement de type (*type erasure*) ne permet pas d'utiliser le paramètre à l'exécution.

La seconde approche qui consiste à utiliser le mécanisme d'introspection de Java possède les caractéristiques inverses. Dans la mesure où il s'agit d'un objet, sa valeur n'est pas connue et ne peut donc pas être prise en compte lors de la compilation. Lors de l'exécution, un tel objet peut toutefois être utilisé pour vérifier qu'un objet est bien du type spécifier ou pour en créer de nouvelles instances.

Ces deux approches sont donc complémentaires et suivant les cas, il conviendra d'adopter l'une, l'autre ou la combinaison des deux. Pour supporter la première approche, chaque paramètre générique de la classe Java qui constitue le composant LLCM$_j$ constitue aussi un paramètre générique du composant. Pour la seconde approche, il faut noter que le paramètre est alors un objet, c'est à dire une valeur de donnée et on utilisera alors la même approche que pour cette dernière catégorie de paramètres.

Finalement, la dernière catégorie de paramètres génériques est constituée de ceux qui représentent une valeur de donnée. Dans la mesure où leur valeur est connue lors de la construction de l'instance, ils se prêtent bien à un passage sous la forme de paramètres du constructeur. C'est cette approche qui est adoptée et chaque paramètre du constructeur de la classe Java qui constitue le composant LLCM$_j$ constitue aussi un paramètre générique du composant.

Interactions Quelles que soient les interactions, elle se traduisent au niveau des composants primitifs en Java par des échanges de référence sur des objets entre les composants. Les ports de LLCM$_j$ sont donc constitués par des accesseurs à des références de la classe qui met en œuvre le composant. Pour les distinguer des méthodes qui ne constituent pas un port, ces accesseurs sont marqués par des annotations.

Deux types de ports sont au moins nécessaires, un premier pour obtenir une référence sur une valeur interne au composant et un second pour spécifier la valeur que doit prendre un élément interne. Ces deux types de ports correspondent en LLCM$_j$ à une méthode publique exposée par la classe qui met en œuvre le composant. Pour le premier cas, cette méthode n'accepte pas de paramètre et sa valeur de retour correspond à la référence exposée par le composant. Une telle méthode doit toujours renvoyer une référence sur le même objet et être marquée par l'annotation `@LlcmjProvide`. Pour le second cas, la méthode ne revoie aucune valeur mais accepte un unique paramètre qui corresponde à la référence importée par le composant. Une telle méthode est assurée de n'être appelée qu'une fois et doit être marquée par l'annotation `@LlcmjUse`.

Ce mécanisme de bas niveau permet de supporter une interaction par appel de méthodes entre composants. Le composant qui souhaite faire appel à des méthodes importe une référence typée par l'interface objet qui spécifie les méthodes devant être mises en œuvre. Le composant qui met en œuvre ces méthodes expose une référence sur l'objet Java qui les met effectivement en œuvre ; il s'agit souvent du même objet que celui qui met en œuvre le composant et qui expose les ports comme c'est le cas pour l'exemple présenté sur la figure 6.5

Une seconde utilisation de ce mécanisme consiste à supporter des attributs de configu-

```
@LlcmjComponent public class WorldGreeter implements Greeter
{
  @LlcmjProvide public Greeter greetWorld() {
    return this;
  }

  @Override public void greet() {
    System.out.println("Hello, world!");
  }
}
```

FIGURE 6.5 – Exemple de composant LLCM$_j$ `WorldGreeter` qui expose un port `greetWorld` typé par l'interface objet `Greeter` qui comporte une unique méthode : `greet`.

ration. Pour cela, le composant spécifie dans sont interface qu'il importe une référence sur un objet typée par un type concret (comme par exemple `String` ou `Integer`) plutôt qu'une interface. La valeur peut alors être fournie par un autre composant ou bien plus probablement lors de la phase d'assemblage pour configurer le composant.

Ces mécanismes permettent de mettre deux composants en relation par des appels de méthodes et de configurer les composants avec des valeurs spécifiées dans l'assemblage ou bien exportées par d'autres composants. Rien n'empêche la valeur exportée par un composant d'être utilisée par plusieurs autres composants. À l'inverse, la question de l'approche à adopter pour permettre à un composant d'importer un ensemble de valeurs à la manière du `uses` *multiple* dans CCM se pose.

De multiples approches peuvent être envisagées pour définir la valeur d'une collection de références comme par exemple :

1. appeler à de multiples reprises une méthode donnée avec en paramètre l'une des références ;
2. appeler une seule fois une méthode avec en paramètre un tableau ou une collection JAVA regroupant les différentes références ; ou
3. remplir une collection exposée par le composant avec les différentes références.

D'un point de vue technique, la solution 2 implique que le choix de la mise en œuvre de la collection n'est pas du ressort du composant. Différent composants peuvent cependant avoir des utilisations particulières de cette collection auxquelles sont adaptées des mises en œuvres spécifiques. Il est donc intéressant de permettre aux choisir la mise en œuvre de la collection par le composant lui-même, ce qui conduit à repousser cette solution.

Les solutions 1 et 3 sont équivalentes de ce point de vue. On peut toutefois noter que dans l'éventualité d'une évolution vers un modèle dynamique, l'approche 3 offre l'avantage de supporter l'ensemble des méthodes nécessaires à la modification de la collection alors que l'approche 1 nécessiterait l'ajout de méthodes supplémentaires au composant. C'est donc l'approche qui consiste à exposer une référence sur une collection depuis le composant et à la remplir pendant la phase de configuration qui est adoptée.

Il faut cependant noter que l'utilisation de l'annotation `@LlcmjProvide` pour exposer la collection ne correspondrait alors plus à la sémantique décrite. Il y a en effet une différence au niveau du contrat des ports `LlcmjProvide` et de ceux utilisés pour l'utilisation d'une collection de références. Dans le cas des premiers, la référence n'est accédée que pour être passée à d'autre composants lors de la phase de configuration alors que pour les second, l'objet référencé (la collection) est modifié. Une nouvelle annotation est donc ajoutée pour marquer ces ports : `@LlcmjMultipleUse`.

6.2.3 Conclusion

Le modèle de composants LLCM$_j$ est un modèle de composants natif local à un processus qui supporte la généricité. Il propose comme seul moyen d'interaction la passage de référence sur un objet JAVA entre les composants. Ce passage de référence se fait depuis un port `LlcmjProvide` vers un port `LlcmjUse` ou bien `LlcmjMultipleUse` dans le cas où plusieurs telles références peuvent être acceptées.

La sémantique des références ainsi passées n'est pas définie par le modèle. Cet aspect fait qu'il offre un faible niveau d'abstraction et qu'il est bien adapté à l'utilisation comme modèle sous-jacent de HLCM. Cette couche additionnelle permet effectivement d'associer des informations sémantiques supplémentaires aux différentes interaction.

Dans le cadre de la mise en œuvre d'une spécialisation de HLCM et en particulier pour la mise en œuvre de la spécialisation basée sur LLCM$_j$, un problème d'initialisation peut toutefois apparaître. En effet, pour exécuter une application basée sur HLCM et LLCM$_j$ il est nécessaire d'exécuter une transformation au déploiement. Si cette transformation est mise en œuvre qui s'appuie sur HLCM et LLCM$_j$ on se trouve en présence d'un cycle de dépendances.

Pour cette raison, les assemblages décrits dans ce chapitre ne s'appuient pas sur HLCM. À la place, ils sont décrits dans une application en JAVA qui commence par créer les diverses instances de composant, puis à les connecter à l'aide de l'API dédiée.

6.3 Les composants de HLCM$_i$

Les mises en œuvres de spécialisations de HLCM peuvent être comparées à des compilateurs : à partir d'un ensemble de fichiers qui décrivent une application de manière abstraite, elles doivent générer une description adaptée aux ressources d'exécution utilisées. Leur exécution se distingue donc en trois phases principales :

1. dans un premier temps les informations sont récupérées, c'est à dire que les fichiers qui décrivent l'application sont lus pour en construire une représentation interne ;
2. ensuite les informations sont traitées, c'est à dire que cette représentation est transformée pour générer une description de l'application spécifiques aux ressources d'exécution ;
3. finalement le résultat est exporté, généralement la description résultante est écrite dans un ou des fichiers mais il peut aussi s'agir d'exécuter directement l'application.

En terme d'approche basée sur les modèles, la première étape construit une représentation interne d'un modèle à partir d'une représentation textuelle. La seconde étape transforme ce modèle en un modèle adapté aux ressources d'exécution. En adoptant la terminologie proposée dans le cadre du MDA [47] par l'OMG, il s'agit de transformer le modèle indépendant de la plateforme (*Platform Independant Model* – PIM) vers un modèle spécifique à la plateforme (*Platform Specific Model* – PSM). Finalement, lors de la dernière étape, le modèle résultant est utilisé pour exécuter l'application, soit directement soit en générant des fichiers qui seront utilisés par le moteur d'exécution.

En plus de posséder des architectures similaires, les mises en œuvres des diverses spécialisations de HLCM partagent aussi de nombreux éléments communs au sein de chacune de ces étapes. On peut par exemple citer l'analyseur des fichiers HLCM dont la syntaxe est indépendante de la spécialisation utilisée ou la mise en œuvre de l'algorithme de transformation décrit en 5.2.3. Il faut toutefois noter que malgré ces similitudes, de nombreuses variations peuvent apparaître entre diverses mises en œuvres de HLCM.

Une première source de variations est constituée des différences entre les diverses spécialisations au niveau du modèle de composants. Tout d'abord, les éléments primitifs de chaque modèle sont différents et les formats de fichiers utilisés pour les décrire le sont aussi. De la même manière, les contraintes qui peuvent être spécifiées sur les mises en œuvres de types génériques peuvent aussi différer et leur sémantique doit être prise en

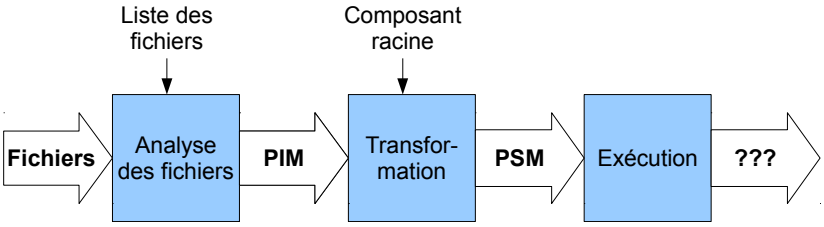

FIGURE 6.6 – Représentation de l'organisation des composants qui forment la colonne vertébrale des mises en œuvres de spécialisations de HLCM.

compte lors de la transformation. Finalement, la phase d'utilisation du résultat dépend fortement du modèle primitif utilisé puisqu'il s'agit globalement d'exporter une application décrite dans ce modèle.

Une seconde source de variations est constituée des divers éléments non centraux à HLCM pour lesquels le choix de l'approche a été laissé ouvert. C'est par exemple le cas du modèle à utiliser pour la description des ressources d'exécution et celui pour décrire le placement des instances de composant sur celles-ci. De la même manière, l'algorithme à utiliser pour choisir la mise en œuvre à utiliser pour les composants et les connecteurs lorsque plusieurs sont disponibles n'a pas été spécifié.

Comme nous l'avons déjà remarqué, ces propriétés font de HLCM un candidat intéressant à une mise en œuvre à base de composants. Il s'agit de décrire chaque mise en œuvre d'une spécialisation de HLCM sous la forme d'un assemblage de composants. Parmi ceux-ci, certains peuvent être commun aux diverses mises en œuvres alors que d'autres peuvent être spécifiques à une mise en œuvre particulière. Dans ce cadre, on désigne sous le nom de HLCM$_i$ l'ensemble des composants utilisés au sein des mises en œuvre de spécifications de HLCM.

6.3.1 Architecture

L'architecture en trois étapes qui a été identifiée correspond globalement à un modèle de flux de travail. Chaque étape s'appuie sur des données construites à l'étape précédente pour construire de nouvelles données mises à disposition de l'étape suivante. Les données qui transitent entre chaque étape sont des modèles décrits sous la forme d'instances de classes JAVA. En plus de ces informations internes, des informations additionnelles doivent êtres fournies à chaque étape. Pour créer le modèle indépendant de la plateforme, il est nécessaire de spécifier les fichier à analyser et lors de l'opération de transformation il est nécessaire de spécifier le composant à instancier. Une vue à haut niveau des composants utilisés pour la mise en œuvre de cet enchaînement et des relations entre leurs instances est présentée sur le schéma de la figure 6.6.

Un premier choix concerne l'approche pour la transmission des modèles entre ces différentes étapes. Il s'agit de transmettre une référence sur l'objet à la racine de l'arbre qui sous-tend le modèle. On peut cependant noter que dans le cadre qui nous intéresse l'enchaînement n'est exécuté qu'une seule fois.

Une première approche pourrait consister à transmettre directement cette référence à l'aide du mécanisme dédié de LLCM$_j$. Un composant exporterait la référence à l'aide d'un port `LlcmjProvide` et les autres l'importeraient à l'aide d'un port `LlcmjUse`. Ainsi ils manipulerait tous le même objet. Cette approche a l'inconvénient d'imposer à l'objet partagé d'exister dès la configuration des composants. Ce n'est pas forcément possible puisque chaque modèle est a priori créé au cours de l'exécution.

Une seconde approche consiste à s'appuyer sur une interaction par appel de méthode. Une méthode dédiée peut alors être utilisée pour passer une référence sur la racine de l'arbre. Cette approche permet de passer la référence sur l'objet qui met en œuvre la méthode dès la phase de configuration des composants LLCM$_j$ mais de n'appeler la méthode que

quand on sait que l'objet a été créé. Deux variations autour de cette approche sont possibles :
- soit c'est le composant qui construit le modèle qui l'envoie comme paramètre d'une méthode à ceux qui l'utilisent, on parle d'approche *push* ;
- soit c'est les composants qui utilisent le modèle qui vont le récupérer comme est valeur de retour d'une méthode de celui qui l'a construit, on parle d'approche *pull*.

Dans la mesure où chaque étape ne s'exécute qu'une fois et qu'il n'y a donc pas d'exécution concurrente, les deux approches sont à peu près équivalentes en terme de complexité. L'approche *pull* a l'avantage de ne pas nécessiter de port multiple, et c'est donc celle qui a été choisie.

En plus des composants qui forment ces trois étapes, il est nécessaire de développer un composant qui organise leur exécution. Il s'agit d'exécuter les différentes phases dans l'ordre en leur fournissant les différentes options de configurations requises : liste des fichiers en entrée, composant à instancier, répertoire de sortie, etc. Ces informations pouvant varier avec les spécificités des diverses étapes, le composant doit cependant être spécifique à chaque mise en œuvre.

6.3.2 Les modèles

Le modèle de HLCM Le modèle de HLCM a été décrit en Ecore comme spécifié dans les chapitres 3, 4 et 5. Cette description a été effectuée en utilisant la syntaxe offerte par Emfatic et a été séparée en deux parties. Une première partie comporte l'ensemble des classes qui ne sont pas spécifiques aux modèles de composants, c'est à dire la modélisation des types de données, de leur valeurs et des concepts qui concernent la généricité. Cette modélisation comporte 39 classes décrites en 135 lignes de Emfatic. La seconde partie modélise les aspects spécifiques à un modèle de composants en adoptant l'approche proposée pour HLCM. Elle comporte 88 classes décrites à l'aide de 315 lignes de Emfatic. Au total, c'est donc environs 470 lignes de Emfatic qui sont nécessaires. Une fois traduites dans la représentation sous forme de classes Java, ces deux parties comportent respectivement 6400 et 18300 lignes de code [1].

Il faut toutefois noter qu'en raison de l'approche pragmatique adoptée pour EMF, certains aspects du modèle ne peuvent pas être décrits en Ecore. C'est par exemple le cas lorsque la valeur d'un attribut d'une classe est dérivée de la valeur d'autres attributs. Il est alors possible de spécifier en Ecore que l'attribut est dérivé et qu'il n'est pas nécessaire de stocker sa valeur dans la représentation Java. Il n'est cependant pas possible de décrire la logique qui permet d'obtenir la valeur de l'attribut dans le modèle Ecore. C'est au sein de sa représentation Java que le comportement doit être spécifié en mettant en œuvres certaines méthodes qui sont laissées vides par la phase de génération.

L'approche normalement utilisée consiste à effectuer cette mise en œuvre puis à spécifier à l'aide d'annotations Java que la méthode a été modifiée par l'utilisateur pour qu'elle ne soit pas écrasée lors d'éventuelles nouvelles générations du code. Cette approche a toutefois l'inconvénient de nécessiter que l'ensemble des classes Java générées soient inclues dans le gestionnaire de versions. Elle empêche aussi de supprimer les classes existantes avant de nouvelles générations. Dans le cas où le nom de certaines classes a changé au sein du modèle, les classes Java qui correspondaient à l'ancien nom restent alors et posent problème.

Afin de contourner ces problèmes, nous avons fait le choix de plutôt hériter des classes générées et de surcharger les méthodes qui doivent être mises en œuvre à la main. Cette approche permet de garder l'ensemble du code mis en œuvre manuellement complètement distinct du celui qui est générée. Grâce à l'utilisation du modèle de conception *fabrique* dans le code généré, il est aisé de remplacer toutes les instanciations de classes du modèle par les classes codées à la main qui en héritent lorsque c'est nécessaire. Ce code écrit à la main

1. L'estimation du nombre de lignes de code des différentes parties de la mise en œuvre a été effectuée à l'aide de l'outil SLOCCount développé par David A. Wheeler qui permet d'ignorer automatiquement les lignes non significatives. Cet outil ne possède toutefois pas de connaissance du langage Emfatic. La faible taille de ces fichiers a cependant permis d'effectuer une estimation manuelle pour ceux-ci.

comporte 350 lignes de Java pour la partie indépendante du concept de composant et 1200 pour la partie spécifique à HLCM.

En plus de ce code purement fonctionnel, du code qui facilite l'accès au modèle dans diverses situations fréquentes a été développé. Il s'agit par exemple de retrouver l'objet qui modélise la valeur d'un paramètre pour une référence générique étant donné son nom. Ce code totalise 445 lignes de Java réparties en 180 lignes spécifiques à HLCM et 268 qui en sont indépendantes.

Le modèle spécifique à la plateforme Le modèle spécifique à la plateforme est utilisé pour décrire le résultat de l'opération de transformation. Son métamodèle ne décrit pas comme dans le cas du modèle indépendant de la plateforme des types associés à des mises en œuvre. Dans la mesure où tous les choix ont été effectuées, il est possible de connaître le contenu de chaque instance, et ce sont donc celles-ci qui y sont modélisées.

Pour simplifier le modèle en évitant de devoir y recopier toutes les informations issues du type et de la mise en œuvre pour chaque instance, ce modèle dépend du modèle indépendant de la plateforme. Chaque instance y référence le type et la mise en œuvre qui ont donné lieu à sa création. Cette approche permet notamment de ne pas nécessiter d'étendre ce modèle pour chaque spécialisation de HLCM, les informations spécifiques à la spécialisation étant cantonnées au modèle indépendant de la plateforme.

Il faut toutefois noter qu'en plus d'être utilisé pour décrire l'application spécifique à la plateforme, ce métamodèle est utilisé pour décrire l'application en cours de transformation. Il comporte donc un ensemble de classes qui n'apparaissent plus une fois la transformation terminée qui représentent par exemple un composant dont la mise en œuvre n'a pas encore été choisie.

Ce modèle comporte 41 classes dont la description en Emfatic nécessite environs 160 lignes de code. Une fois traduites dans la représentation sous forme de classes Java, cette description comporte environs 1500 lignes de code. De plus, 800 lignes de Java écrites à la main au sein de classes Java qui héritent de celles générées décrivent des aspects du modèle qui ne peuvent pas être décrits en Ecore.

6.3.3 Analyse des fichiers

L'analyseur de fichiers HLCM est développé en utilisant l'environnement Xtext. La description de sa grammaire augmentée des informations nécessaire à la construction du modèle nécessite environs 400 lignes.

Il faut toutefois noter que cette approche implique une corrélation forte entre le modèle et la syntaxe textuelle. Dans la majorité des cas, la syntaxe abstraite correspond effectivement à la syntaxe textuelle. Il existe cependant des limitations dans Xtext qui font que ça ne peut pas toujours être le cas.

C'est par exemple le cas pour les arguments de types génériques. Les arguments qui attribuent une valeur à un paramètre qui représente un composant ou ceux qui attribuent une valeur à un paramètre qui représente un connecteur sont modélisés par deux classes différentes : `ComponentArgument` et `ConnectorArgument`. En terme de syntaxe, les deux cas correspondent simplement à une chaîne de caractère qui représente le nom du type et d'éventuels paramètres. Dans la mesure où l'analyseur de fichier s'appuie sur une grammaire hors-contexte, il n'est pas possible de distinguer ces deux cas. Pour gérer ces situations, six classes additionnelles au modèle de HLCM ont été décrites au sein d'un modèle destiné à coller de plus près à la syntaxe de HLCM pour être utilisées quand de telles ambiguïtés apparaissent.

Une première opération de transformation de modèle est ensuite appliquée pour remplacer leurs instances par des instances des classes plus spécifiques lorsque le contexte est connu et qu'il devient possible de distinguer les cas. Cette opération de transformation est effectuée par un composant dédié et elle nécessite environs 430 lignes de Java.

6.3.4 L'opération de transformation

Algorithme de transformation L'opération de transformation est mise en œuvre au sein d'un composant LLCM$_j$ qui accède à une instance du PIM pour construire une instance du PSM. Ce composant met en œuvre l'algorithme présenté au chapitre 5. Il faut cependant noter que l'algorithme ne spécifie pas comment le choix entres les différentes mises en œuvres utilisables pour une instance donnée doit être effectué. Afin de permettre de multiples mises en œuvres de ces choix, le composant externalise cet aspect en faisant appel à une interface dédiée importée au travers d'un port.

Cette interface comporte deux méthodes, la première est utilisée lorsque c'est la mise en œuvre d'instances de composants qu'il faut choisir et l'autre pour les connexions. Afin de permettre au composant qui effectue le choix d'avoir un maximum d'information à disposition, l'algorithme développe l'assemblage autant que possible avant de faire une requête de choix. Les méthodes de l'interface acceptent un unique paramètre qui est une collection d'instance de composants (ou de connexions pour la seconde) dont la mise en œuvre n'a pas encore été choisie. Le composant qui effectue le choix doit choisir une mise en œuvre pour au moins une des instances de l'ensemble qui lui est passé. Il spécifie ce choix en modifiant l'instance pour lui associer une mise en œuvre.

Cette mise en œuvre comprend environs 2000 lignes de codes répartie à peu près équitablement au sein de deux classes. La première constitue la mise en œuvre effective du composant et comporte en interne la mise en œuvre de l'algorithme proprement dit. La seconde décrit le contexte de la transformation qui doit être stocké dans une pile pour gérer la généricité. Il s'agit de sauvegarder les associations entre les paramètres génériques et leur valeur dans un contexte donné. Cette classe met aussi en œuvre une interface qui facilite l'empilement de ces contextes et leur utilisation pour requérir la valeur actuelle d'un paramètre donné.

Choix Une version naïve et indépendante de la spécialisation de l'algorithme de choix a aussi été développée. Il s'agit de parcourir les mises en œuvres disponibles et d'utiliser la première valide rencontrée. Pour qu'une mise en œuvre donnée soit valide, il faut qu'elle mette en œuvre le type de l'instance étudiée et que les contraintes qu'elle spécifie soient remplies.

Ces conditions doivent être remplies quel que soit l'algorithme de choix utilisé. Leur mise en œuvre est donc effectué au sein d'un composant distinct pour permettre de partager ce code. Vérifier qu'un type mis en œuvre correspond à celui d'une instance particulière peut être fait indépendamment de la spécialisation, mais les contraintes qui peuvent être appliquées dépendent de la spécialisation. Pour maximiser la réutilisation du code, chaque contrainte est gérée par un composant distinct et la comparaison des types dans un autre.

Ces composants mettent en œuvre deux interfaces distinctes, une pour les composants et une pour les générateurs. Chacune de ces interfaces comporte une unique méthode qui prend en paramètre un objet qui représente le type de l'instance à mettre en œuvre et un autre qui représente la mise en œuvre dont la validité est testée. Cette méthode renvoie une valeur booléenne qui est à faux si la mise en œuvre ne peut pas être utilisée. Dans le cas contraire, les paramètres libres de cette mise en œuvre peuvent être contraints pour correspondre à l'instance mise en œuvre.

Validité de type Pour vérifier que deux spécialisations de type génériques sont identiques, il faut s'assurer que les types eux-mêmes sont les mêmes et que les valeurs de leurs paramètres correspondent aussi. Puisque ces derniers peuvent à leur tour être générique, le même processus doit être appliqué et ce sont finalement des arbres qui doivent être mis en correspondance.

Cependant une mise en œuvre générique peut elle-même posséder des paramètres génériques comme sur l'exemple de la figure 6.7. Dans ce cas, ces paramètres sont libres, c'est à dire qu'il est de la responsabilité de l'algorithme de choix de choisir leur valeur. Cependant,

```
component MyComponent<porttype P1, porttype P2> exposes {
  ...
}

composite MyImplementation<porttype P> implements MyComponent<P,P> {
  ...
}
```

FIGURE 6.7 – Un type de composant `MyComponent` et une mise en œuvre `MyImplementation`. Cette mise en œuvre comporte un paramètres dont la valeur doit être choisie lors de la phase de transformation. Ce choix n'est pas arbitraire car dans ce cas, la valeur du paramètre `P` doit être la même que celle des arguments passé explicitement par l'utilisateur pour les paramètres du type de composant. Cette mise en œuvre ne peut pas non plus être utilisée dans toute les situations, elle n'est utilisable que dans le cas où les paramètres `P1` et `P2` de `MyComponent` ont la même valeur.

cette valeur doit être compatible avec les utilisations qui en sont faite, en particulier comme argument du type mis en œuvre.

Il est aussi possible qu'un même paramètre de la mise en œuvre soit utilisé comme valeur pour plusieurs paramètres du type mis en œuvre. Dans ce cas, il faut que ces paramètres aient effectivement la même valeur. Finalement, c'est un algorithme de mise en correspondance de deux arbres comportant éventuellement des variables libres qui doit être mis en œuvre. Cet algorithme correspond à un cas particulier de l'algorithme d'unification de termes par exemple utilisé dans PROLOG. Dans ce cas, l'un des deux arbres à unifier ne comporte que des constantes. Ce cas particulier simplifie un peu l'algorithme qui est déjà relativement simple à mettre en œuvre dans le cas général.

Une rapide analyse des mises en œuvre existantes a montré qu'elles impliquent des dépendances importantes et auraient certainement nécessité le développement autant de code pour les intégrer dans HLCM$_i$ que ce qui est nécessaire à une mise en œuvre complète de l'algorithme. Le choix a donc été effectue de développer la mise en œuvre de cet algorithme d'unification sans s'appuyer sur une bibliothèque existante. Le composant de vérification de la validité de type des mises en œuvres qui comporte cet algorithme nécessite finalement un peu moins de 900 lignes de code JAVA.

6.4 Les composants spécifiques à HLCM$_i$/CCM

6.4.1 Fichiers CCM étendu

Afin de gérer la spécialisation HLCM/CCM, il est nécessaire de lire non seulement les fichiers HLCM proprement dit, mais aussi les fichiers écrits en IDL CCM étendu qui décrivent les composants primitifs. Pour utiliser l'outil XTEXT dans ce but, il est nécessaire de construire un métamodèle qui représente la syntaxe abstraite de ces fichiers et d'en décrire une grammaire et sa mise en relation avec le modèle. L'analyseur de fichiers IDL étendus a été construit en développant d'abord un analyseur pour la syntaxe IDL non étendue en s'appuyant sur la grammaire définie dans le standard OMG.

Analyse de fichiers IDL CCM non étendu La grammaire fournie par l'OMG a été modifiée en renommant les règles pour correspondre aux conventions pour les classes ECORE. De plus, chaque appel d'une règle dans la partie droite d'une autre a été affecté à un attribut pour obtenir une grammaire utilisable par XTEXT. Une fonctionnalité de XTEXT qui permet d'inférer un métamodèle ECORE à partir de la grammaire a été utilisée. La grammaire et le métamodèle ont ensuite été légèrement retravaillés pour unifier les éléments qui présentent

des points communs. Des classes dont les différents éléments qui possèdent des caractéristiques communes héritent ont été crées.

Il faut cependant noter que la grammaire de l'IDL CCM ne constitue pas une spécification complète du contenu potentiel des fichiers. Ceux-ci doivent en effet être préalablement traités par un préprocesseur compatible avec la norme C. C'est notamment par ce préprocesseur que sont géré les mécanismes d'inclusion de fichiers en utilisant le mot clef `#include`.

Une première approche pour gérer cet aspect pourrait consister à utiliser un préprocesseur existant qui se conforme à cette norme comme celui distribué avec GCC par exemple puis à utiliser le fichier généré par cet outil. En cas d'erreur de syntaxe, ce choix rend cependant complexe le rapport d'erreur à l'utilisateur puisque le fichier et la ligne auxquelles l'erreur a été détectée ne sont pas connues (ou doivent être reconstruites depuis des informations ajoutées au fichier). De plus, cette approche s'intègre mal avec les outils d'Eclipse et rend notamment très difficile l'obtention d'un éditeur éditeur intégré qui est normalement automatique.

La solution qui a été adoptée consiste donc à gérer les inclusions directement dans la grammaire et à ignorer les autres commandes préprocesseur comme s'il s'agissait de commentaires. Après avoir ainsi créé un analyseur de fichiers CCM purs, une seconde étape a consisté à étendre cette grammaire pour analyser les fichiers qui comportent les extensions propres à HLCM/CCM définies au chapitre 5.

Analyse des fichiers IDL étendus pour HLCM/CCM Au niveau de la syntaxe abstraire, les extensions pour HLCM/CCM constituent à faire le lien entre les composants CCM et les composants HLCM/CCM qu'ils mettent en œuvre ainsi que le lien entre les ports CCM et ceux de HLCM/CCM. Ces modification consistent à ajouter aux classes qui modélisent les composants et les ports des attributs qui référencent un composant ou une connexion respectivement dans le modèle de HLCM. Finalement le métamodèle de CCM étendu comporte 77 classes Ecore décrites en environs 320 lignes de Emfatic qui transformées en Java correspondent à environs 15000 lignes de code.

En ce qui concerne l'analyseur de fichiers, une difficulté apparaît du fait que le choix a été fait de décrire les annotations dans des commentaires CCM affin d'assurer la compatibilité avec les outils existant. En effet, dans la grammaire, les commentaires sont simplement ignorés à l'étape de l'analyse lexical à partir du moment où le marqueur de commentaire (//) apparaît et jusqu'à la fin de la ligne. Puisque des éléments significatifs peuvent dorénavant apparaître sur ces lignes, la grammaire a été modifiée pour que les lignes qui comportent un arobase juste après le marqueur de commentaire ne soient pas considérées comme des commentaires.

Ce changement implique cependant que la fin d'une annotation ne peut pas être marquée par une fin de ligne qui ne peut plus être différenciée d'un autre caractère d'espacement après la phase d'analyse lexicale. Il en résulte qu'il est possible pour ce parseur de mettre l'annotation sur la même ligne que le mot clef qui la suit. Il faut cependant faire attention au fait que les outils existant ignorent de telles lignes entièrement et que de tels fichiers seraient alors incompatibles avec les outils existant.

Une fois ces modifications faites, l'analyse de fichiers IDL étendue génère une instance d'un métamodèle dédié qui correspond globalement à celui de CCM étendu de quelques informations qui mettent en correspondance ces éléments avec ceux de HLCM. Ce modèle ne peut cependant pas être utilisé tel quel lors de la phase de transformation qui attend un modèle dans lequel les éléments primitifs sont modélisés par des classes qui héritent de celles présentes dans le métamodèle HLCM. Il est donc nécessaire d'étendre ce métamodèle et de mettre en œuvre une pré-transformation qui adapte le modèle issue de l'analyse des fichiers en un modèle utilisable par l'opération de transformation.

Extension du modèle de HLCM Du point de vue du métamodèle de HLCM, les extensions nécessaires consistent à décrire une classe par élément pouvant être spécifié dans les

fichiers IDL et manipulé dans les fichiers HLCM. Il existe trois tels éléments : les types de données, les événements et les composants CCM dont la description nécessite 20 de lignes de EMFATIC qui correspondent en JAVA à environs 700 lignes de code.

Ces classes sont simplement construites en héritant des classes modélisant les éléments primitifs dans HLCM et en leur ajoutant des références sur les objets du modèle de l'IDL CCM étendu. L'opération de génération des objets de ces classes est relativement directe et a été mise en œuvre en environ 150 lignes de JAVA.

Il faut noter qu'en plus de ces classes spécifiques, il existe dans HLCM/CCM un ensemble d'instances prédéfinies de types de ports, connecteurs et générateurs primitifs. Ces instances sont stockées dans un fichier au format XMI pour être automatiquement chargées quand HLCM$_i$/CCM est utilisé.

6.4.2 Opération de choix

Un second point qui a dû être étudié pour HLCM/CCM concerne l'opération de choix. En effet, la mise en œuvre naïve indépendante de la spécialisation présentée en 6.3.4 ne prend pas en compte les ressources d'exécution et ne fournit pas de placement des composants. Cette solution ne permet pas de prendre en compte les contraintes de placement qui peuvent être spécifiées dans HLCM/CCM.

Modèle de ressources matérielles Un métamodèle pour la description des ressources d'exécution a donc été créé ainsi qu'un métamodèle pour la description de l'association des instances de composant dans le modèle spécifique à la plateforme à ces ressources. Le métamodèle pour la description des ressources reprend celui qui a été proposé dans le cadre d'ADAGE [38]. Il comporte 25 classes décrites en environs 70 lignes d'EMFATIC qui une fois transformées en JAVA correspondent à environs 3200 lignes de code.

Il faut cependant noter que ce modèle est relativement complexe ce qui rend son utilisation dans l'algorithme de choix malaisé. Afin de simplifier les requêtes les plus fréquentes, aussi bien pour accéder au contenu du modèle que pour le modifier, des méthodes d'accès ont donc été développées au sein de composants dédiés. Ces composants accèdent au modèle au travers d'un port et proposent des opérations de manipulation en offrant la possibilité d'effectuer des appels de méthodes à l'aide d'interfaces dédie. Deux tels composants ont été mis en œuvre, le premier pour les accès qui ne modifient pas le modèle et le second pour ceux qui le modifient en respectivement 280 et 150 lignes de JAVA.

Pour construire ce modèle, plusieurs sources peuvent être utilisées. Deux composants qui correspondent à deux sources différentes ont pour l'instant été mis en œuvre. Le premier construit un modèle fixe destiné à effectuer des tests dans un environnement connus. En s'appuyant sur l'interface d'accès et de modification du modèle, il peut être réduit au minimum et ne nécessite qu'une cinquantaine de lignes de JAVA.

Le second analyse des fichiers existant qui décrivent la plateforme expérimentale *Grid'5000* et les filtre en fonction des ressources effectivement réservées. Dans la mesure où ces fichiers sont au format XML, une approche aurait pu constituer à créer un métamodèle et à s'appuyer sur une opération de transformation de modèle. Cette approche possède cependant le désavantage de nécessiter une transformation de modèle qui n'est pas simple à coder. Dans la mesure où des méthodes d'accès facilitant la modification du modèle sont disponible, le choix a été fait d'utiliser les outils d'analyse XML classique pour analyser les fichiers et de construire le modèle à la main. Ce code nécessite environs 150 lignes de code JAVA.

Un second métamodèle est aussi nécessaire pour associer les instances de composant du modèle spécifique à la plateforme aux ressources sur lesquelles elles doivent s'exécuter. Ce modèle est très simple puisqu'il se réduit à trois classes décrites en une quinzaine de lignes d'EMFATIC. La version JAVA de ce métamodèle comporte environs 900 lignes de code.

Choix avec prise en compte des ressources La prise en compte des ressources matérielles dans l'algorithme de choix a été faite de manière minimale. Il s'agit en effet simplement de réutiliser l'algorithme naïf présenté en 6.3.4 et qui consiste à choisir la première mise en œuvre acceptable pour les composants et les connecteurs. Afin d'assurer que les contraintes de placement qui peuvent être spécifiées pour ces mises en œuvres soient respectées, un filtre additionnel est développé.

Le filtre constitue un composant similaire à celui qui a été présenté pour assurer que les mises en œuvres soient valide en ce qui concerne le type mis en œuvre. Il prend en paramètre une mise en œuvre et vérifie que les contraintes qui y sont spécifiées sont respectées, si ce n'est pas le cas, la mise en œuvre est considérée invalide.

Une fois l'ensemble des mises en œuvre choisies, les instances de composant sont placées selon un algorithme tourniquet qui distribue ces instances sur les ressources d'exécution. Dans le cas où une instance est liée par une contrainte de localisation dans le même processus avec d'autres instances, c'est l'ensemble du groupe d'instance qui est ainsi placé.

6.4.3 Génération des fichiers finaux

Le dernier élément spécifique qui a dû être mis en œuvre dans le cadre de HLCM$_i$/CCM est un composant qui permette d'exécuter l'application CCM correspondant à l'assemblage spécifique à la plate-forme généré. Ce composant s'appuie donc sur le PSM mais aussi sur la description des ressources et sur le plan qui permet de déterminer sur quelles ressources devra s'exécuter chaque instance de composant.

Il génère un assemblage de composants CCM sous la forme d'un fichier descripteur d'assemblage de composants (*Component Assembly Descriptor* – CAD). Il génère aussi un fichier destiné à ADAGE et décrivant le déploiement qui doit être effectué pour les instances décrites dans ce CAD.

Ces deux types de fichiers s'appuient sur un formalisme XML. Le choix a de nouveau été fait de s'appuyer sur les mécanismes classique de manipulation en utilisant le modèle objet du document (*Document Object Model* – DOM) plutôt que sur une transformation de modèle.

Le CAD comporte trois sections qu'on peut définir de manière simplifiée comme suit :

- la première définit l'ensemble des types de composant utilisés au sein de l'assemblage ;
- la seconde définit l'ensemble des instances de composants et certaines contraintes de collocation ; et
- la troisième décrit les connexion entre les instances.

Pour générer ce fichier, l'algorithme appliqué consiste simplement à parcourir toutes les instances primitives de composant présentes dans le modèle final. Ce parcours s'appuie sur le modèle de plan de déploiement et est effectué processus par processus. Pour chaque processus, un groupe de collocation est créé dans la seconde section du fichier CAD. Pour chaque instance de composant située dans ce processus, une instance est décrite à l'intérieur du groupe de collocation. Les types de l'ensemble des instances rencontrées sont sauvegardés dans une collection JAVA qui supprime les doublons.

Une fois l'ensemble des instances de composant parcouru, la collection qui regroupe leurs types est utilisée pour remplir la première section du fichier CAD. Finalement, l'ensemble des connexions primitives est parcouru pour remplir la troisième section du fichier.

Concernant le fichier destiné à ADAGE, il s'agit simplement d'y spécifier les contraintes de placement décrites dans le modèle de plan de déploiement. Dans la mesure où ce modèle est très fortement inspiré de celui d'ADAGE, il s'agit simplement de le parcourir pour l'exprimer dans un autre format.

A final, la mise en œuvre de ces composants nécessite moins de 250 lignes de JAVA au total.

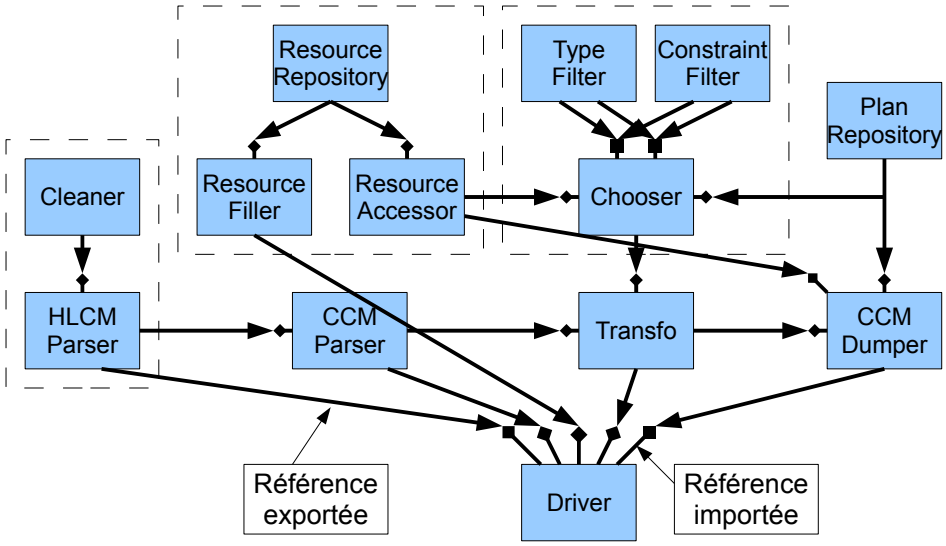

FIGURE 6.8 – Présentation de l'architecture de HLCM$_i$/CCM en terme d'assemblage de composants LLCM$_j$. Les rectangles en tirets représentent des regroupements logiques d'instances qui constitueraient de bon candidats à la création de composite dans un modèle comportant ce concept.

6.4.4 Conclusion

En plus de ces composants fonctionnels, un composant qui gère le parcours de la ligne de commande et qui pilote l'exécution est nécessaire. Dans le cas de HLCM$_i$/CCM, ce composant comporte environs 100 lignes de code JAVA. Il s'agit du composant `Driver` présenté sur le schéma de la figure 6.8 qui décrit l'architecture complète de HLCM$_i$/CCM.

Un premier composant qui est exécuté par le pilote est l'analyseur de fichiers HLCM (`HLCM Parser`). Il construit une représentation interne du modèle qui est décrit dans les fichiers dont le pilote lui passe les noms. Une fois ce modèle construit, il le transmet à un composant qui «nettoie» ce modèle (`Cleaner`), c'est à dire qu'il y supprime les éléments présents uniquement à cause des limitations de XTEXT. Il rend ensuite ce modèle disponible au travers d'une interface dédiée.

Le second composant exécuté par le pilote est l'analyseur de fichiers CCM étendus (`CCM Parser`). Il s'appuie sur le modèle construit par l'analyseur de fichiers HLCM. Il construit un second modèle —sur-ensemble du premier— qui comporte les mises en œuvres primitives CCM qu'il a trouvé dans les fichiers dont le pilote lui a passé le nom. Ce second modèle (le PIM) est à nouveau exposé par le composant à l'aide d'une interface dédiée.

Le troisième composant exécuté par le pilote est le générateur de ressources matérielles (`Resource Filler`). Il utilise l'interface d'accès en écriture offerte par le dépôt de ressources matérielles (`Resource Repository`) pour y créer la description des ressources *Grid'5000* qu'il lit dans un fichier XML. L'accès à ce modèle de ressources en lecture avec une interface simplifiée est proposé par le composant d'accès aux ressources (`Resource Accessor`) qui s'appuie pour ça sur le modèle exposé par le dépôt.

Une fois l'ensemble de ces informations récupérées, le pilote lance l'opération de transformation à proprement parler en spécifiant le type de composant HLCM à instancier. La mise en œuvre de cette transformation au sein du composant dédié (`Transfo`) utilise le PIM exposé par l'analyseur de fichiers CCM et expose une fois la transformation terminée le PSM. Il utilise aussi le service de choix de mise en œuvre offert par le composant dédié (`Chooser`).

Le composant de choix accède en lecture au modèle des ressources matérielles et en écriture au plan de déploiement exposé par le dépôt de plan (`Plan Repository`). Il utilise deux filtres pour éliminer les mises en œuvre invalides : un filtre de type (`Type filter`) et

un filtre de contraintes (`Constraint Filter`).

Finalement, la dernière étape exécutée par le pilote est gérée par le composant d'écriture de fichiers finaux CCM (`CCM Dumper`). Ce composant accède à trois modèles : le PSM, la description des ressources matérielles et le plan de déploiement. Il génère alors les fichiers CCM et ADAGE nécessaire à l'exécution de l'application dans le répertoire qui lui est spécifié par le pilote.

6.5 Conclusion

Cette mise en œuvre permet de démontrer qu'il est possible de mettre en œuvre l'ensemble des concepts proposés au sein de HLCM et en particulier de sa spécialisation HLCM/CCM. Il faut noter qu'un élément de cette mise en œuvre justifie particulièrement son statut de prototype. Il s'agit des algorithmes de choix de mises en œuvre utilisés.

Le choix d'une approche extensible basée sur des composants permet d'envisager sans difficulté de proposer des mises en œuvres qui s'appuient sur d'autres algorithmes ou bien d'autres spécialisations de HLCM. C'est notamment le cas des spécialisations basées sur DHICO qui seront présentées avec les éléments nécessaires à leur mise en œuvre dans la section 7.4. Cette section s'attardera notamment sur la question de la mise en œuvre d'algorithmes de choix plus réalistes dans $HLCM_i$.

L'utilisation de la plateforme de développement proposée au sein du projet ECLIPSE a permis de proposer ces mises en œuvre avec un effort de développement raisonnable. Pour comparaison, un premier effort de développement où les métamodèles étaient directement décrits en JAVA a dû être abandonné en cours de développement après quelques mois à cause de la quantité de code trop importante à écrire. La version s'appuyant sur EMF qui a ensuite été développée en remplacement a pu être amené à parité de fonctionnalité en un temps bien moindre.

Les évolutions récente de la plateforme EMF poussent à penser qu'il devrait devenir possible dans un avenir proche de s'appuyer sur des technologies supplémentaires pour réduire l'effort de développement nécessaire. On peut notamment penser aux langages dédiés aux transformations de modèles.

Finalement, $HLCM_i$ propose une plate-forme qui permet l'évaluation du modèle HLCM lui-même. C'est ce que propose le chapitre suivant.

Chapitre 7

Évaluation

Les chapitres précédents ont présenté l'approche que nous proposons pour définir un modèle de composants logiciels qui permette la définition de nouveaux schémas de composition. Cette approche s'appuie sur les concepts de hiérarchie, de choix de mise en œuvre, de généricité et de connecteurs comme entités de première classe qui ont été regroupés au sein de HLCM. Un objectif poursuivi par la spécification de ce modèle consiste à permettre l'exécution efficace d'applications de calcul scientifique sur des ressources matérielles variées.

Ce chapitre propose une évaluation de ce modèle et en particulier des performances à l'exécution qu'il permet d'atteindre. Pour cela, il présente et évalue les mises en œuvres de plusieurs squelettes et connecteurs au sein de HLCM. Il s'intéresse aussi à la possibilité de s'appuyer sur HLCM pour la mise en œuvre d'heuristiques de choix non triviales.

Dans un premier temps, la section 7.1 revient sur les critères qui avaient été identifiés dans le chapitre 3 pour évaluer les approches pour l'ajout de fonctionnalités dédiées au calcul à haute performance dans les modèles de composants logiciels. Les sections 7.2 et 7.3 s'appuient sur ces critères pour évaluer des mises en œuvre des fonctionnalités qui ont été présentées dans le chapitre 5 : l'utilisation d'une mémoire partagée et d'appels parallèles de méthodes respectivement. La section 7.4 évalue l'utilisation des concepts de HLCM ainsi que de $HLCM_i$ au sein d'un modèle de programmation par composition dont les concepts sont relativement éloignés des modèles de composant classiques : la composition par communications collectives hiérarchies au sein du modèle DiscoGrid.

7.1 Critères d'évaluation

Les chapitres précédents ont introduit le modèle HLCM qui comporte les concepts de choix de mise en œuvre, de hiérarchie, de généricité et de connecteurs comme entité de première classe. L'interface des composants dans HLCM n'est pas formée d'un ensemble de ports comme dans la majorité des modèles de composants logiciels classiques mais s'appuie sur le nouveau concept de connexion ouverte.

Lors d'évaluations préliminaires présentées dans les chapitres 4 et 5, il a été montré :

1. que le concept de généricité utilisé conjointement avec la hiérarchie permet de définir des squelettes algorithmiques au sein du modèle de composants ; et
2. que l'utilisation des concepts de connexion ouverte et d'adaptateur de connexion permettent de s'appuyer sur les connecteur pour spécifier les interactions entre composants même en présence de hiérarchie et de choix de mise en œuvre.

Ces deux aspects forment les deux facettes des fonctionnalités dédiées au calcul à haute performance qui ont été étudiées dans le chapitre 2. Le postulat a été fait dans le chapitre 3 qu'un modèle qui permettrait d'introduire de tels constructions serait particulièrement adapté aux applications de calcul scientifique.

Ce chapitre évalue donc dans quelle mesure ce postulat est valide. Pour cela, il présente les mises en œuvres qui peuvent être fournies pour ces concepts et et évalue leur adéquation avec les critères qui avaient été définis dans le chapitre 3. Pour rappel, ces critères concernaient plusieurs aspects.

Expressivité Un premier aspect concerne l'expressivité du modèle, c'est à dire la capacité d'y décrire des applications qui s'appuient sur les diverses fonctionnalités nécessaires. Cet aspect comporte deux critères d'évaluation principaux.

Un premier critères concerne la capacité du modèle à permettre une mise en œuvre **exhaustive** des fonctionnalités. C'est à dire qu'il devrait permettre d'exprimer l'ensemble des fonctionnalités nécessaires dans les application visées. Même si les fonctionnalités qui ont été présentées au chapitre 2 ne constituent pas l'intégralité des fonctionnalités possibles, on peut considérer qu'elles offrent une base intéressante pour cette évaluation.

Un autre critère concerne la possibilité de décrire dans le modèle des fonctionnalités qui soient **compatibles** entre elles. Cette compatibilité devrait permettre d'utiliser au sein d'une même application des combinaisons des différentes fonctionnalités aussi complexes qu'il est nécessaire. Elle ne devrait pas nécessiter que les fonctionnalités existantes soient prises en compte au moment de la conception d'une nouvelle fonctionnalité.

Performances Un second aspect pour l'évaluation du modèle concerne les performances qu'il permet d'obtenir pour les applications qui l'utilisent pour leur mise en œuvre. A nouveau, cet aspect comporte deux critères d'évaluation.

Un point concerne les différents **sur-coûts** dûs à l'utilisation du modèle par rapport à une solution qui serait mise en œuvre sans celui-ci. Ces sur-coûts peuvent être observés lors de l'exécution à proprement parler, par exemple si le modèle impose de s'appuyer sur des appels indirects de méthodes à l'aide de pointeurs plutôt que sur des appels résolus à l'édition de lien. Ces sur-coûts peuvent aussi se retrouver à d'autres étapes du processus, comme par exemple lors du déploiement de l'application.

Un second point concerne la **portabilité** des applications décrites à l'aide du modèle sur différentes ressources d'exécution. En effet, nous avons vu que la recherche de puissance de calcul amène à utiliser une grande variété d'architectures matérielles. Dans ce contexte, les performances d'une application dépendent de son adaptation aux ressources matérielles effectivement utilisées. Il devrait être possible de tirer parti de manière efficace de ces diverses ressources y compris dans le cas où elles sont créées après l'application elle-même.

```
connector SharedMem<role access>;
```

FIGURE 7.1 – Déclaration du connecteur `SharedMem` comportant un unique *rôle* : `access`.

Interface d'utilisation Finalement, le troisième aspect dans l'évaluation du modèle concerne sa facilité d'utilisation. Cet aspect est certainement le plus subjectif à juger, mais on peut en tirer au moins deux critères.

Un premier critère concerne la capacité du modèle à assurer l'aspect **boîte noire** des composants. C'est à dire qu'il devrait être possible d'utiliser un composant de manière optimale sans nécessiter une compréhension profonde de son fonctionnement interne.

Un second point concerne la possibilité d'offrir une **séparation des préoccupations** entre le code métier et le code de support. Ces deux aspects sont en effet du domaine de compétences de personnes différentes. Le code métier est *a priori* de la responsabilité des experts du domaine étudié alors que le code de support est plutôt de la responsabilité des experts de l'ingénierie logicielle ou bien du matériel sur lequel s'exécute l'application.

7.2 Couplage par partage de mémoire

L'exemple du couplage de code par mémoire partagée a été présenté en 5.1.2. Cet exemple est constitué de deux composants possédant chacun une mise en œuvre séquentielle et une mise en œuvre parallèle qui interagissent au travers d'une mémoire partagée. Il a ensuite été montré en 5.3.2 qu'il était possible de s'appuyer sur HLCM/CCM pour développer un connecteur qui soit utilisé quelles que soient les mises en œuvres des composants.

7.2.1 Description

Ce connecteur appelé `SharedMem` est présenté sur la figure 7.1. Il comporte un unique rôle nommé `access` qui est rempli par chaque instance primitive de composants qui accède à la mémoire. Dans le cas des mises en œuvres séquentielles, ce sont deux instances primitives qui remplissent ce rôle, mais dans le cas où des mises en œuvre parallèles sont utilisées, il peut s'agir d'un nombre quelconque d'instances de composants.

Pour accéder à cette mémoire partagée, les composants s'appuient sur un service spécifié par l'interface IDL `DataAccess` présentée sur la figure 7.2. Ce service comporte deux aspects :

- il permet d'associer un identifiant à des zones d'un espace mémoire (logiquement) partagé et d'accéder à une zone étant donné son identifiant ;
- il permet de manipuler les verrous d'exclusion mutuelle implicitement associés à chacune de ces zones.

Cette interface met cependant en avant deux limitations du modèle HLCM/CCM. Tout d'abord, elle s'appuie sur le type opaque `datapointer` utilisé pour transmettre un pointeur sur des données. Ce type d'interaction n'est possible qu'au sein d'un même espace d'adressage et n'est donc pas prévu dans CORBA. Les types de port `LocalFacet` et `LocalReceptacle` ont dû être introduits pour contourner cette limitation. Dans HLCM/CCM, une contrainte de localisation dans le même processus est imposée pour des instances de composants qui interagissent par une connexion `UseProvide` dont un rôle est rempli par un port de l'un de ces deux types.

Deuxièmement, cette interface offre un faible niveau d'abstraction puisqu'elle offre une vision de la mémoire partagée correspondant à un tableau d'octets. Une approche qui permettrait de spécifier un type pour les données partagées nécessiterait que l'interface soit générique, or la notion de généricité n'existe pas dans CORBA.

Comme cela a été montré au chapitre 5, une interactions par partage de mémoire peut se ramener à un ensemble d'instances primitives de composant qui participent à une unique connexion. Cette possibilité est offerte grâce au concept de connexion ouverte qui permet de

```
interface DataAccess
{
    // création d'une zone mémoire avec identifiant
    void share(in unsigned long size, out datapointer data, in string id);

    // récupération des informations sur une zone mémoire associée à un identifiant
    void access(in string id, out datapointer data, out unsigned long size);

    // découplage d'une zone mémoire et de son identifiant
    void detach(in string id);

    // manipulation des verrous associés aux zones mémoire
    void write_lock(in string id);
    void read_lock(in string id);
    void release(in string id);
};
```

FIGURE 7.2 – Déclaration en IDL OMG de l'interface `DataAccess`.

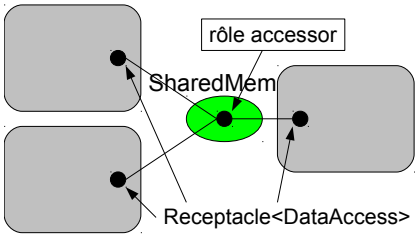

FIGURE 7.3 – Assemblage intermédiaire où tous les composants participent à une même connexion indépendamment de leur distribution initiale au sein de la hiérarchie.

s'affranchir de la distribution des instances primitives dans la hiérarchie des composants. C'est cet assemblage intermédiaire qui est représenté par le schéma de la figure 7.3.

La suite de cette section présente trois mises en œuvre du connecteur `SharedMem` quand son rôle `access` est rempli par des ports de type `LocalReceptacle<DataAccess>`. Ces mises en œuvre correspondent à trois distributions des instances de composant qui accèdent à la mémoire :

- quand les instances se situent toutes dans le même espace d'adressage ;
- quand elles ont toutes accès au même système d'exploitation offrant les mécanismes de partage de mémoire Posix ; et
- quand elles sont distribuées sur des machines distinctes connectées par un réseau.

7.2.2 Mémoire physiquement partagée

Dans le cas où le matériel fournit une mémoire physiquement partagée et que les différentes instances de composants connectées se situent dans le même espace mémoire, la mise en œuvre peut être réduite à son minimum. Il n'est pas nécessaire de gérer la cohérence des données entre les composants, puisque c'est le matériel qui s'en charge. Les verrous offerts par le système d'exploitation peuvent être utilisés pour l'exclusion mutuelle.

Il est ainsi simplement nécessaire de gérer l'association d'un identifiant et d'un verrou d'exclusion mutuelle aux zones mémoire. Ces services peuvent être offerts par une unique instance de composant. C'est cette mise en œuvre centralisée que propose un premier générateur dont la description en HLCM est présentée sur la figure 7.4

Ce connecteur délègue l'ensemble des appels au composant `LocalMemoryStore`. Ce composant possède une mise en œuvre primitive qui s'appuie sur les mécanismes d'allocation

```
generator LocalSharedMem<Integer N>
    implements SharedMem<access=each(i:[1..N]){
        LocalReceptacle<DataAccess>}>
{
  LocalMemoryStore<N> store;
  each(i:[1..N]){store.access[i].user+=access[i];}
}
```

FIGURE 7.4 – Définition en HLCM du générateur `LocalSharedMem` qui met en œuvre le connecteur `SharedMem`. Il s'appuie sur le composant `LocalMemoryStore` pour gérer un nombre quelconque de clients dans le même espace d'adressage.

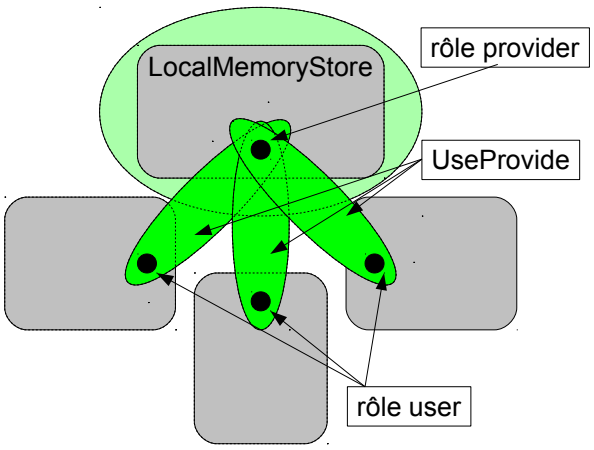

FIGURE 7.5 – Assemblage de la figure 7.3 après l'application du générateur `LocalSharedMem`.

mémoire et d'exclusion standards offerts par le système.

Dans cette mise en œuvre, il n'est pas nécessaire de spécifier de contrainte explicite concernant la collocation des composants qui accèdent à la mémoire partagée. En effet, chaque connexion `UseProvide` entre les instances de composant qui accèdent à la mémoire et l'instance `store` imposent une contrainte de collocation. Par transitivité, la contrainte de collocation entre l'ensemble des instances de composant est donc implicitement spécifiée.

Appliqué à l'assemblage présenté sur le schéma de la figure 7.3, ce générateur donne lieu à un assemblage tel que celui présenté sur le schéma de la figure 7.5.

7.2.3 Mémoire partagée Posix

Dans le cas où les instances de composant ne sont pas situées dans le même espace d'adressage, cette première mise en œuvre n'est pas utilisable. Dans le cas où ils se situent sur une unique machine et accèdent au même espace physique de mémoire, il existe cependant des mécanismes offerts par le système d'exploitation pour partager de la mémoire. C'est le cas de la mémoire partagée Posix qui permet de projeter un même espace mémoire dans deux espaces d'adressage distincts (éventuellement à des adresses différentes).

Dans ce cas, il est nécessaire que le générateur comporte plusieurs instances de composant afin qu'une instance se situe dans chaque espace d'adressage d'où une instance de composant accède à la mémoire. Dans la mesure où l'interface offerte par la mémoire partagée Posix est très proche de celle définie par `DataAccess`, il n'est nécessaire pour ces instances que d'effectuer la traduction d'interface.

C'est cette approche qu'adopte le générateur `PosixSharedMem` qui est présenté sur la figure 7.6. Ce générateur s'appuie sur le composant `PosixSharer` dont une instance est associée à chaque utilisateur. Une mise en œuvre primitive de ce composant s'appuie sur

```
generator PosixSharedMem<Integer N>
    implements SharedMem<access=each(i:[1..N]){LocalReceptacle<DataAccess>}>
    when samesystem(each(i:[1..N]){this.access})
{
  each(i:[1..N]){
    PosixSharer node[i];
    node[i].access.user += this.access[i];
  }
}
```

FIGURE 7.6 – Définition en HLCM du générateur `PosixSharedMem` qui met en œuvre le connecteur `SharedMem`. Il s'appuie sur autant d'instances du composant `PosixSharer` que d'instances qui accèdent à la mémoire partagée. Il n'est utilisable que lorsque les instances de composant sont situées sur la même machine.

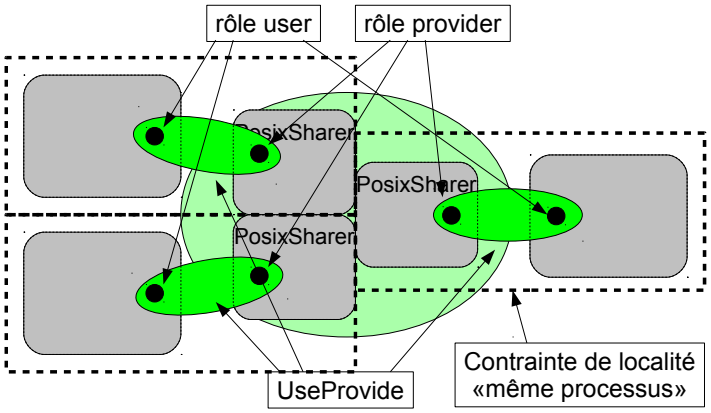

FIGURE 7.7 – Assemblage de la figure 7.3 après l'application du générateur `PosixSharedMem`.

le mécanisme de mémoire partagée Posix dans lequel il partage notamment les verrous d'exclusion mutuelle.

Cette mise en œuvre n'est utilisable que quand l'ensemble des instances de composant ainsi connectées s'exécutent au sein du même système d'exploitation, ce qui est assuré par la contrainte `samesystem` de la définition de la figure 7.6. Chaque instance est assurée d'être située dans le même processus que l'instance qui lui fournit le service grâce à la contrainte implicite portée par la connexion `UseProvide` offrant le service `DataAccess`.

Appliqué à l'assemblage présenté sur le schéma de la figure 7.3, ce générateur donne lieu à un assemblage tel que celui présenté sur le schéma de la figure 7.7.

7.2.4 Système distribué de mémoire partagée

Finalement, la dernière situation est celle où les instances de composant qui accèdent à la mémoire partagée sont distribuées sur plusieurs machines s'appuyant sur des mémoires physiques différentes. Dans ce cas, il est nécessaire de s'appuyer sur un mécanisme distribué de mémoire partagée comme par exemple JUXMEM [5].

JUXMEM est une solution pair à pair où chaque pair supporte une partie de l'espace mémoire et offre l'accès à l'espace mémoire logiquement partagé. Certains aspects comme les méta-données ou l'initialisation sont toutefois gérés de manière centralisée par un *manager*. Plusieurs *managers* peuvent être utilisés mais dans le cadre de nos expérimentations, nous nous sommes restreint au cas où une unique instance est utilisée.

Le générateur `JuxMem` présenté sur la figure 7.8 s'appuie sur ce mécanisme. Deux types de composant y sont utilisés : `JuxMemPeer` qui encapsule un pair et `JuxMemManager` qui encapsule

```
generator JuxMem<Integer N>
    implements SharedMem<access=each(i:[1..N]){LocalReceptacle<DataAccess>}>
{
  JuxMemManager<N> manager;
  each(i:[1..N]){
    JuxMemPeer peer[i];
    peer[i].access.user += access[i];
    merge({peer[i].internal, manager.internal[i]});
  }
}
```

FIGURE 7.8 – Définition en HLCM du générateur `DistributedSharedMem` qui met en œuvre le connecteur `SharedMem`. Il s'appuie sur autant d'instances du composant `JuxMemPeer` que d'instances qui accèdent à la mémoire partagée. Ces instances sont toutes connectées à une instance de `JuxMemManager` qui assure leur initialisation et leur connexion.

```
interface InterPeer
{
  void config(out string host, out unsigned short port, out unsigned short id);
};
```

FIGURE 7.9 – Déclaration en IDL OMG de l'interface `InterPeer`. Elle ne comporte qu'une unique méthode permettant aux pairs JUXMEM de récupérer l'adresse réseau ainsi que le port où écoute le *manager* JUXMEM ainsi que de s'octroyer un identifiant unique.

le *manager*. Une instance de `JuxMemPeer` est créée par instance de composant qui accède à la mémoire ainsi qu'une unique instance de `JuxMemManager`.

Dans JUXMEM, les communications entre les pairs et avec le *manager* s'appuient sur des *sockets* réseau du système d'exploitation. Chaque pair obtient les adresses des autres pairs depuis le *manager*. Il est cependant nécessaire de fournir l'adresse du *manager* à chaque pair pour initialiser le processus. C'est le rôle de la connexion de chaque pair avec le *manager*.

Ces connexions sont de type `UseProvide`. Le composant *manager* offre le service composé d'une unique méthode qui permet de renvoyer les informations nécessaires à la connexion.

Il n'est pas ici nécessaire de spécifier de contrainte concernant le placement des instances de composant qui participent à la connexion. Dans la mesure où chaque instance de composant qui accède à la mémoire logiquement partagée s'appuie sur l'interface `DataAccess` utilisée au travers d'un port de type `LocalReceptacle`

Appliqué à l'assemblage présenté sur le schéma de la figure 7.3, ce générateur donne lieu à un assemblage tel que celui présenté sur le schéma de la figure 7.10.

7.2.5 Analyse

Ces trois mises en œuvres permettent de proposer une première évaluation de certains aspects de HLCM et plus particulièrement de sa spécialisation HLCM/CCM. On s'appuie pour cela sur les critères qui ont été rappelés dans la section 7.1.

Expressivité En terme d'expressivité tout d'abord, on peut noter que si HLCM/CCM a permis la spécification des ces trois mise en œuvre du partage de mémoire, cela s'est fait au prix de quelques entorses au modèle.

Une première entorse concerne l'interface dédiée au partage de données. Il a en effet été nécessaire d'ajouter à la spécialisation de HLCM des types de ports permettant d'assurer que les composants qui interagissent au travers d'une connexion donnée soient situés dans

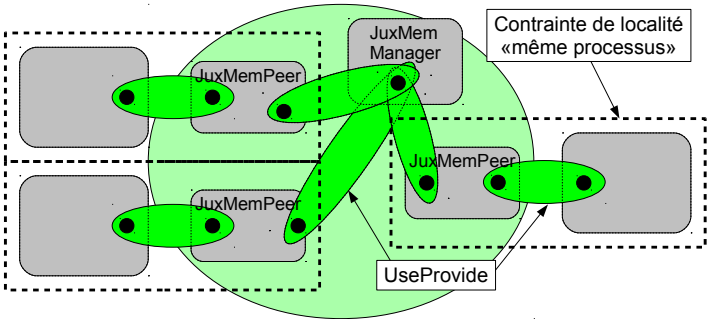

FIGURE 7.10 – Assemblage de la figure 7.3 après l'application du générateur `JuxMem`.

le même espace d'adressage. Cette modification est en contradiction avec le but que nous avions défini de pouvoir mettre en œuvres des fonctionnalités additionnelles sans modification du modèle.

Une seconde entorse concerne les interactions entre instances de composant qui ne sont pas spécifiées dans le modèle. Dans le cas de la mise en œuvre basée sur le mécanisme Posix par exemple, cette interaction n'est pas spécifiée entre les instances de composant du générateur `PosixSharedMem`. De manière similaire, dans le cas de la mise en œuvre basée sur JUXMEM, les interactions au travers de *sockets* ne sont pas spécifiées.

On peut considérer que ces entorses sont dues au fait que le modèle de composant sous-jacent utilisé (CCM) n'est pas le plus adapté. Il impose en effet une approche pour les interactions des composants (appels de méthodes s'appuyant sur CORBA) qui fait sens pour assurer la compatibilité dans le cadre d'un modèle de composant classique mais qui s'avère être limité dans le cadre de HLCM.

Pour permettre de s'appuyer entièrement sur le modèle pour les interactions, il pourrait être envisagé de s'appuyer sur un modèle sous-jacent qui expose l'ensemble des moyens d'interaction offerts par le système. C'est par exemple ce que tente de proposer le langage de description d'architecture UNICON [63] avec des connecteurs primitifs pour décrire les interactions au travers de la mémoire, de fichiers, de *pipes*, ou d'appel de procédures par exemple. Cette approche empêcherait cependant de s'appuyer sur du code existant par exemple dans le cas de JUXMEM et nécessiterait de que l'intégralité de la pile logiciel depuis le système d'exploitation jusqu'aux applications utilisateurs s'appuient sur HLCM.

Une seconde option plus réaliste consisterait donc à proposer une spécialisation de HLCM qui s'appuie sur un modèle sous-jacent dans lequel des générateurs primitifs peuvent être décrits à la manière des mises en œuvre de composants. Un tel modèle permettrait ainsi de prendre en compte les interactions offertes par des codes existant sans en nécessiter de modification importante.

Performances En ce qui concerne les performances, on peut tout d'abord constater qu'en raison de la modification qui a été apporté à HLCM/CCM pour permettre le passage de pointeurs sur de la mémoire, HLCM/CCM n'intervient en rien pour les accès aux données eux-mêmes. Il permet simplement de mettre en place un partage qui est ensuite géré par les outils sur lesquels s'appuient les mises en œuvres. Des évaluations précises des performances des différentes solutions n'ont donc pas été conduites par manque de temps et parce qu'elles auraient principalement permis d'exhiber les performances des mécanismes sous-jacents.

On peut cependant noter que HLCM permet le choix de la mise en œuvre à utiliser en fonction du matériel sur lequel est déployé l'application. Le résultat est qu'il est possible de s'appuyer sur des mécanismes performants quand les instances de composant sont déployées localement tout en conservant la possibilité d'être portable sur des architectures matérielles plus compliqués.

Un autre type de sur-coûts introduits par HLCM sont ceux qui sont liés à l'opération

```
interface Service
{
    void apply(in Vector v);
};
```

FIGURE 7.11 – Déclaration en IDL OMG de l'interface `Service` qui comporte une unique méthode `apply` acceptant un vecteur `v` comme seul paramètre.

de transformation au déploiement de l'application. Pour évaluer ces sur-coûts, divers assemblages ont été transformés pour évaluer la durée de cette opération. Ces assemblages comportent des composants parallèles avec des degrés de parallélisme allant jusqu'à mille et en utilisant pour leurs interactions par mémoire partagées les trois mises en œuvres présentées dans cette section. Leur transformation ne nécessite pas plus de quelques secondes (environs neuf secondes dans le pire cas). Dans le cadre de déploiement d'applications de cet ordre de grandeur sur des ressources distribuées, ce temps peut cependant être considéré comme négligeable.

Lorsqu'on analyse ce temps de transformation, on se rend compte qu'il est dominé par un temps fixe pour l'initialisation de l'infrastructure JAVA, l'enregistrement des divers modèles et l'analyse des fichiers qui nécessite environs 5 secondes. Le temps additionnel correspond à la transformation elle-même. Il faut cependant noter que l'algorithme de choix qui a été utilisé est l'algorithme naïf qui choisit la première mise en œuvre utilisable. Dans le cadre d'une utilisation réelle, il faudrait prendre en compte le temps d'exécution des heuristiques de choix. La section 7.4 reviendra sur cet aspect.

Interface d'utilisation L'interface d'utilisation pour partager de la mémoire entre composants est relativement aisée à comprendre. L'accès ne se fait que par des connexions ouvertes de type dédiée ce qui assure l'aspect «boîte noire». L'aspect technique du partage de données est clairement isolé dans les générateurs pour laisser le code qui l'utilise se concentrer sur son utilisation.

Au sein des générateurs eux même cependant, comme cela a déjà été identifié, les composants ne respectent pas entièrement le concept de «boîte noire». Certaines interactions ne sont pas prises en compte par le modèle. Nous avons vu que cet aspect pourrait être corrigé par l'utilisation d'un modèle sous-jacent plus adapté.

7.3 Couplage par appel de méthode

L'exemple du couplage par appel de méthodes de code séquentiels ou parallèles a été présenté en 5.1.2. Cet exemple est constitué de deux composants possédant chacun une mise en œuvre séquentielle et une mise en œuvre parallèle qui interagissent par des appels de méthodes. Il a ensuite été montré en 5.3.3 qu'il était possible de s'appuyer sur HLCM/CCM pour rendre les composants parallèles compatibles avec des appels de méthode séquentiels. Il a aussi été montré que cette possibilité ne se faisait pas au détriment de la possibilité de proposer des mises en œuvres optimisées pour le cas où les deux composants sont parallèles.

Dans le cadre de cette évaluation, on s'appuiera sur l'exemple d'interactions par appel de méthodes à l'aide du service très simple décrit par l'interface `Service` présentée sur la figure 7.11 qui ne comporte qu'une unique méthode avec un unique paramètre. Ce service est utilisé uniquement à des fins d'évaluation de performances et il n'applique donc aucun traitement aux données.

Le type de composant `ServiceProvider` dont la définition est présentée sur la figure 7.12 correspond au fournisseur du service dans l'assemblage à haut niveau de deux composants. Un type de composant réciproque `ServiceUser` correspond à l'utilisateur du service.

```
component ServiceProvider exposes
{
  UseProvide<provider={Facet<Service>}> offeredService;
}
```

FIGURE 7.12 – Définition du composant `ServiceProvider` en HLCM. Il expose une connexion ouverte `offeredService` de type `UseProvide` dont le rôle `provider` est rempli par un port de type `Facet<Service>`.

Dans la mesure où chacun de ces composants correspondent à l'utilisation ou à la mise en œuvre d'un unique service, il est aisé d'en proposer une mise en œuvre séquentielle primitive. Leur interaction par une connexion de type `UseProvide` où le rôle `user` est rempli par un port de type `Receptacle<Service>` et le rôle `provider` par un port de type `Facet<Service>` correspond à un générateur primitifs, c'est à dire qu'il est géré directement par le modèle sous-jacent CCM.

7.3.1 Composants parallèles

Dans le cas d'une mise en œuvre parallèle de ce composant, il est nécessaire de s'appuyer sur un composant interne qui est répliqué. Ces composants internes doivent offrir un service spécifiée par une interface qui est liée à l'interface `Service` mais diffère. En effet, les données doivent être distribuées entre les différentes instances qui forment le composite. Un premier point qui doit être étudié concerne donc l'approche à adopter pour spécifier cette distribution.

Spécification de la distribution Spécifier la distribution des données comporte deux aspects. Un premier point concerne le type de distribution adoptée. Par exemple, une matrice peut être répliquée sur chaque instance, distribuée par bloc, par ligne, par colonne ou par l'une des variantes cycliques de ces distributions. Un second point concerne la partie des données ainsi découpées qui doit être transmise à chaque instance.

Deux approches peuvent être adoptées au sein de HLCM/CCM pour spécifier cette distribution :

- dans une approche statique, c'est l'interface du service exposé par ces instances qui comporte les informations permettant de déterminer quelle partie des données leur est destinée ;
- dans une approche plus dynamique, ces instances effectuent à l'exécution une opération de configuration qui leur permet de spécifier les données qu'elles attendent.

L'option statique nécessite que chaque instance expose une interface différente puisque chacune doit recevoir une partie différente des données. Pour décrire une distribution du vecteur par bloc cycliques par exemple, il est nécessaire de connaître la taille des blocs et l'indice de ceux qui sont destinés à chaque instances. Ces deux dernières informations peuvent être déduites du nombre total d'instances dans le composant et du rang de l'instance concernée. Un exemple d'interface qui permettrait de spécifier ces informations est présenté sur la figure 7.13.

Cette interface s'appuie cependant fortement sur la généricité qui n'est pas présente dans CORBA. Il ne s'agit donc pas d'une interface IDL valide. On pourrait envisager de supprimer le paramètre qui spécifie la taille des blocs et de s'appuyer sur une taille fixée par le type lui même au prix d'une moindre réutilisation de code. Les deux autres paramètres sont liés au degré de parallélisme du composant parallèle. Appliquer la même solution reviendrait donc à fixer ce degré de parallélisme en diminuerait fortement la possibilité de s'adapter aux ressources.

L'option dynamique consiste transmettre les informations concernant les paramètres de la distributions au travers d'appels de méthode plutôt que par des paramètres générique. Le

```
// N = Nombre de processus dans le composant parallèle
// R = Rang dans cet ensemble de processus
// S = Taille des blocs en nombre d'éléments
interface StaticPartialService<Integer N, Integer R, Integer S>
{
    void apply(in VectorBlocs<N,R,S> v);
};
```

FIGURE 7.13 – Déclaration dans un langage inspiré de l'IDL OMG de l'interface `StaticPartialService` qui comporte l'ensemble des informations pour la distribution des données.

```
interface PartialService
{
    void apply(in VectorBlocs v);
};
```

FIGURE 7.14 – Déclaration en IDL OMG de l'interface `PartialService` qui spécifie le type de distribution utilisée (`VectorBlocs`) mais pas ses paramètres.

composant peut alors offrir une version simplifiée du service principal qui ne comporte pas ces informations comme spécifié par exemple par l'interface `PartialService` présentée sur la figure 7.14. Concernant le service qui sert à transmettre les paramètres de la distribution, l'initiative de l'appel de méthode peut être aux instances qui mettent en œuvre le service ou bien à celles qui l'utilisent. Dans la mesure où il ne semble pas exister d'argument favorisant l'une ou l'autre de ces solutions, la figure 7.15 présente une interface que doivent appeler les instances lors de la phase de configuration pour transmettre leurs paramètres.

Dans la mesure où avec cette approche, les paramètres de distribution ne sont connus qu'à l'exécution, ils ne peuvent notamment pas être utilisés dans les divers choix opérés par HLCM lors du déploiement. C'est particulièrement problématique dans le cas favorable où les deux composants parallèles utilisent exactement la même distribution des données. Il serait en effet intéressant de pouvoir détecter ce cas pour ne pas instancier toute l'infrastructure de redistribution des données.

Dans les cadre des expériences qui ont été effectuées, l'absence de généricité au sein de CCM a été le principal facteur de choix entre ces deux approches. C'est la raison pour laquelle c'est la version dynamique qui a été adoptée.

Composant interne Le type de composant qui correspond aux instances internes du composant parallèle s'appuie donc sur deux services distincts. Le service spécifié par l'interface `PartialService` est mis en œuvre par le composant. Le service spécifié par l'interface `DataDistribution` est utilisée par le composant. Ces deux services sont cependant fortement corrélés et ne peuvent pas être utilisés l'un sans l'autre. Ils sont donc regroupés au sein d'un *bundle* `ProviderPartialServiceFacet` dont la définition est présentée sur la figure 7.16. Un *bundle* `ProviderPartialServiceReceptacle` correspond à l'interface inverse.

```
interface DataDistribution
{
    //distribution du paramètre "v" de la méthode "apply"
    void apply_v(in long nb_instances, in long rank, in long block_size);
};
```

FIGURE 7.15 – Déclaration en IDL OMG de l'interface `DataDistribution` qui spécifie les paramètres de la distribution du vecteur `v` de la méthode `apply`.

```
bundletype ProviderPartialServiceFacet {
  UseProvide<provider={Facet<PartialService>}> service;
  UseProvide<user={Receptacle<DataDistribution>}> distribution;
}
```

FIGURE 7.16 – Définition du *bundle* `ProviderPartialServiceFacet`. Il comporte une connexion ouverte `service` dans laquelle un port de type `Facet<PartialService>` remplit le rôle `provider`. Il comporte aussi une connexion ouverte `distribution` dans laquelle un port de type `Receptacle<DataDistribution>` remplit le rôle `user`.

```
generator ProviderPartialServiceUseProvide
  implements UseProvide<user={ProviderPartialServiceReceptacle},
      provider={ProviderPartialServiceFacet}>
{
  merge(this.user.service, this.provider.service);
  merge(this.user.distribution, this.provider.distribution);
}
```

FIGURE 7.17 – Définition en HLCM du générateur `ProviderPartialServiceUseProvide`. Ce générateur met en œuvre les connexions `UseProvide` où un *bundle* `ProviderPartialServiceFacet` remplit le rôle `provide` et un *bundle* `ProviderPartialServiceReceptacle` remplit le rôle `user`. Il fusionne les deux connexions ouvertes qui correspondent au service lui-même ainsi que les deux qui correspondent au transfert des paramètres de distribution.

Afin de permettre le connexion de ces *bundle*, un générateur `ProviderPartialServiceUseProvide` dont la définition est présentée sur la figure 7.17 est utilisé. Ce générateur gère ces connexions en fusionnant simplement les sous connexions qu'elles contiennent deux à deux.

Du côté client, chaque instance utilise aussi le service `DataDistribution` pour décrire la partie des données dont elle est responsable. Le service décrit par l'interface `PartialService` intervient, cependant il s'agit d'un service que les instances utilisent à l'inverse du côté serveur. Les deux ports y ont donc la même direction (ils correspondent tous les deux à des services utilisés par les instances). Leur regroupement ne peut donc pas s'appuyer sur les même *bundles* et deux nouveaux *bundles* sont créés : `UserPartialServiceFacet` et `UserPartialServiceReceptacle`. De la même manière que pour le côté serveur, un générateur dédié `UserPartialServiceUseProvide` est aussi créé. Ces éléments sont similaires à ceux du côté serveur et ne sont donc pas décrits.

C'est cette interface `ProviderPartialServiceFacet` qu'offre le composant `PartialServiceProvider` qui est répliqué au sein du composant parallèle du côté fournisseur du service. La définition de ce composant est décrite sur la figure 7.18 Ce composant est mis en œuvre de manière primitive à l'aide d'un composant CCM. À nouveau, un composant qui propose la réciproque pour le côté client est aussi développé : `PartialServiceUser`.

```
component PartialServiceProvider exposes
{
  UseProvide<provider={ProviderPartialServiceFacet}> partialService;
}
```

FIGURE 7.18 – Définition du composant `PartialServiceProvider` en HLCM. Ce composant expose une connexion ouverte `partialService` de type `UseProvide` dont le rôle `provider` est rempli par un port de type `ProviderPartialServiceFacet`. Il est destiné à être répliqué au sein d'un composant parallèle.

```
bundletype ParallelServiceFacet<Integer N> {
  each(i:[1..N]){ UseProvide<provider={ProviderPartialServiceFacet}> part[i]; }
}
```

FIGURE 7.19 – Définition du *bundle* `ParallelServiceFacet` qui contient `N` connexions `UseProvide` appelées `part` et dont le *rôle* `provider` est rempli par un port de type `ProviderPartialServiceFacet`.

```
composite ParallelServiceProvider<Integer N> implements ServiceProvider
{
  each(i:[1..N]){
    PartialServiceProvider providerPart[i];
  }
  this.offeredService.provider += bundle ParallelServiceFacet<N> {
    each(i:[1..N]){
      part[i] = providerPart[i].partialService;
    }
  }
}
```

FIGURE 7.20 – Définition de la mise en œuvre composite de composant `ParallelServiceProvider` qui possède un paramètre libre entier `N`. Elle met en œuvre le composant `ServiceProvider`. Elle contient `N` instances du composant `PartialServiceProvider` appelées `providerPart`. Le rôle `provider` de la connexion ouverte `offeredService` exposée par le composant est rempli par un *bundle* de type `ParallelServiceFacet` dont les connexions internes `part` sont celles exposées par les instances `providerPart`.

Regroupement des connexions Le composant parallèle comporte donc un ensemble d'instances du type de composant `PartialServiceProvider`. Ces instances exposent chacune une connexion ouverte de type `UseProvide` dont le rôle `provider` est rempli par un port de type `ProviderPartialServiceFacet`. Ces connexions ne peuvent à nouveau pas être utilisées indépendamment ; il est nécessaire de les appeler simultanément avec des données issues du même vecteur pour que la sémantique prévue soit respectée. Elles sont donc regroupées au sein d'un *bundle* `ParallelServiceFacet` dont la définition est présentée sur la figure 7.19.

La mise en œuvre parallèle du composant `ServiceProvider` dont la définition avait été présentée sur la figure 7.12 consiste donc à créer les instances du composant `PartialServiceProvider` et à regrouper leurs connexions ouvertes au sein d'un *bundle*. C'est ce que fait le composite `ParallelServiceProvider` dont la définition est présentée sur la figure 7.20 et dont un exemple d'instance est présenté sur le schéma de la figure 7.21.

L'utilisation d'un *bundle* de type `ParallelServiceFacet` pour remplir le rôle `provider` de la connexion ouverte `offeredService` ne correspond pas au type `Facet<Service>` spécifié dans le type du composant. Pour permettre cette opération il est donc nécessaire de mettre en œuvre un adaptateur de connexion qui rende les deux types compatibles.

7.3.2 Adaptateur de connexion

Le but des adaptateurs de connexion qui doivent être développés est de rendre une connexion `UseProvide` dont le rôle `provider` est rempli par un *bundle* `ParallelServiceFacet` compatible avec une connexion dont ce rôle est rempli par un port `Facet<Service>`. Pour ce faire il est nécessaire de créer un assemblage qui accepte des appels sur un port `Facet<Service>` et qui s'appuie sur les informations fournies par les instances du com-

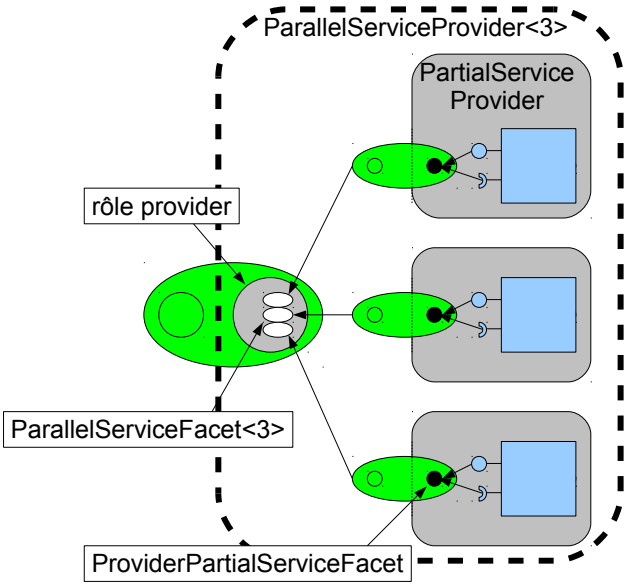

FIGURE 7.21 – Exemple d'instance de `ParallelServiceProvider` avec un degré de réplication de 3. Elle contient 3 instances du composant `PartialServiceProvider`. Les connexions ouvertes qu'elles exposent sont regroupées au sein d'une *bundle* de type `ParallelServiceFacet<3>`. Ce *bundle* remplit le rôle `provider` de la connexion ouverte exposée par le composant.

posant parallèle pour distribuer les données et appeler l'ensemble des services des ports `Facet<PartialService>`.

L'étude de l'environnement PACO++ dédié à la gestion de l'appel parallèle de méthodes permet d'identifier deux approches pour gérer ce cas :

- soit la distribution est gérée du côté du client et les données sont transmises à chaque processus du composant parallèle directement ;
- soit les données sont transmises à un unique processus du composant parallèle qui effectue ensuite la distribution des données.

L'adaptateur de connexion `ServiceScatter` dont la définition est présentée sur la figure 7.22 gère ces deux approches. La redistribution s'effectue là où est localisée l'instance de composant `dist`, soit avec le client, soit avec l'une des instances du composant parallèle. Un exemple d'assemblage issu de l'utilisation de cet adaptateur de connexion est présenté sur la figure 7.23.

Un adaptateur de connexion qui effectue l'inverse est développé pour le cas où c'est le

```
adaptor ServiceScatter<Integer N> supports
    UseProvide<provider={ParallelServiceFacet<N>}> //< supported
    as UseProvide<provider={Facet<Service>}> //< this
{
  Distributor<N> dist;
  merge(this, dist.in);
  each(i:[1..N]){ merge(dist.out[i], supported.provider.part[i]); }
}
```

FIGURE 7.22 – Définition de l'adaptateur de connexion `ServiceScatter` qui supporte une connexion `UseProvide` dont le rôle `provider` est rempli par un port de type `ParallelServiceFacet` comme s'il s'agissait d'une connexion dans laquelle ce rôle est rempli par un port de type `Facet<Service>`. Il s'appuie sur une instance du composant `Distributor` qui gère l'intégralité de la redistribution.

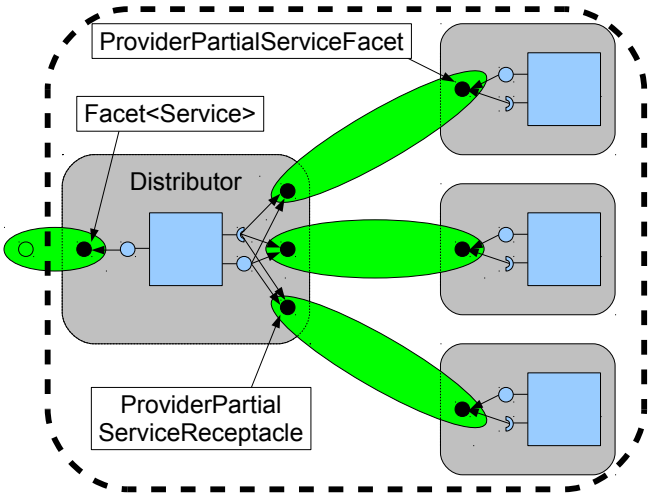

FIGURE 7.23 – Exemple d'utilisation de l'adaptateur de connexion `ServiceScatter` pour rendre le port exposé par la mise en œuvre `ParallelServiceProvider` compatible avec le composant `ServiceProvider`. Cet adaptateur de connexion s'appuie sur une instance du composant `Distributor` qui gère la distribution.

client qui est parallèle : `ServiceGather`. Il permet d'utiliser une connexion `UseProvide` dont le rôle `user` est rempli par un *bundle* `ParallelServiceReceptacle` comme ci celui-ci était rempli par un port `Receptacle<Service>`. Pour cela, il regroupe les données avant d'effectuer l'appel au service. À nouveau, ce regroupement peut se faire au sein de l'un des processus du client parallèle ou bien du côté du serveur.

Ces adaptateurs de connexion permettent de gérer de manière transparente le cas où un client parallèle utilise un service séquentiel. Il permettent de gérer de manière similaire le cas où un client séquentiel utilise un service parallèle. Leur combinaison permet aussi de gérer le cas $M \times N$ où à la fois le client et le serveur sont parallèles. Dans ce cas, le regroupement des données dans un seul processus avant de les redistribuer peut cependant poser un problème de performances. C'est pourquoi des générateurs qui gèrent directement ce cas ont été développés.

7.3.3 Générateurs

Comme pour les adaptateurs de connexions, plusieurs approches peuvent âtre adoptées pour la redistribution des données dans le cas $M \times N$. Une première approche consiste à redistribuer les données par des échanges entre les processus du côté client avant de les envoyer sur les processus du côté serveur auquel elles sont destinées. L'approche opposé est aussi possible, elle consiste à transmettre les données aux processus du côté serveur puis à leur laisser redistribuer les données. Finalement il est possible d'effectuer la redistribution «sur le fil», c'est à dire que chaque processus client envoie les données qui lui sont nécessaires à chaque processus serveur.

Le générateur `MXNServiceUseProvide` dont la définition est présentée sur la figure 7.24 permet ces trois approches. Un assemblage obtenu en utilisant ce générateur dans le cas où `M` vaut deux et `N` vaut trois est présenté sur le schéma de la figure 7.25.

Ce générateur s'appuie sur deux ensemble d'instances de composant :

- les instances `userside` du composant `UsersideRedistributor` —dont la définition est présentée sur la figure 7.26— où la distribution des données est similaire à celle dans le composant parallèle client ; et
- les instances `serverside` du composant `ServersideRedistributor` où la distribution des données est similaire à celle dans le composant parallèle serveur.

```
generator MXNServiceUseProvide<IntegerM, Integer N>
  implements UseProvide<user={ParallelServiceReceptacle<M>},
      provider={ParallelServiceFacet<N>}>
{
  each(i:[1..M]){
    UsersideRedistributor<M,N> userside[i];
    merge({this.user.part[i], userside[i].user});
  }
  merge(each(i:[1..M]){userside[i].collective});
  each(i:[1..N]{
    ServersideRedistributor<M,N> serverside[i];
    merge({this.provider.part[i], serverside[i].provider});
  }
  each(m:[1..M]{
    each(n:[1..N]{
      merge({userside[m].internal[n], serverside[n].internal[m]});    }
  }
}
```

FIGURE 7.24 – Définition en HLCM du générateur `MXNServiceUseProvide` qui met en œuvre le connecteur `UseProvide`. Il s'appuie sur deux types de composants : `UsersideRedistributor` et `ServersideRedistributor`. Les instances du premier sont connectés au instances du composant parallèle client, après avoir reçu les données, elles

Le principe de fonctionnement de ce générateur est que les instances du composant `UsersideRedistributor` commencent par recevoir les données. Ces instances déterminent comment ces données doivent être partitionnées et quelles instances du composant `ServersideRedistributor` doivent les recevoir. Ces dernières attendent d'avoir reçu l'ensemble des données qui leur sont destinées —éventuellement depuis plusieurs sources— avant de les regrouper pour les transmettre aux instances dans le composant parallèle qui met en œuvre le service. Une fois l'appel terminé, les instances du composant `UsersideRedistributor` font appel à une opération de barrière entre elles pour s'assurer que tous les appels sont terminés avant de rendre la main aux instances dans le composant parallèle client.

Pour obtenir les informations sur la distribution des données dans le composant client, les instances `userside` du composant `UsersideRedistributor` s'appuient sur les appels que ceux-ci ont du préalablement effectuer sur l'interface `DataDistribution` au travers la connexion ouverte `user`. De la même manière, pour obtenir les informations sur la distribution des données dans le composant serveur, les instances `serverside` du composant `ServersideRedistributor` s'appuient sur les appels que ceux-ci ont du préalablement effectuer sur l'interface `DataDistribution` au travers la connexion ouverte `provider`.

Les instances `userside` doivent cependant connaître non seulement la distribution du côté client, mais aussi du côté serveur afin de transmettre les bonnes données. C'est la raison pour laquelle les connexions internes s'appuient sur les *bundles* `ProviderPartialServiceReceptacle` et `ProviderPartialServiceFacet` qui regroupent une connexion pour transmettre les données et une connexion pour définir leur distribution.

Les communications collectives entre les instances `userside` se réduisent dans ce cas à l'utilisation de la barrière. La mise en œuvre qui a été proposée est actuellement centralisée et s'appuie sur une unique instance de composant qui attend que l'ensemble des instances aient fait l'appel avant de rendre la main. Le générateur et le composant sur lequel il s'appuie ne sont pas présentés ici car ils ne présentent pas d'intérêts ou de difficultés particuliers.

Comme dans le cas de l'adaptateur de connexion, l'approche utilisée pour la redistribution (côté client, côté serveur ou «sur le fil») dépend de la distribution des instances de composant sur les ressources. Si à la fois les instances `userside` et `serverside` sont créées dans les processus du composant client, c'est là que s'effectue la redistribution. Inverse-

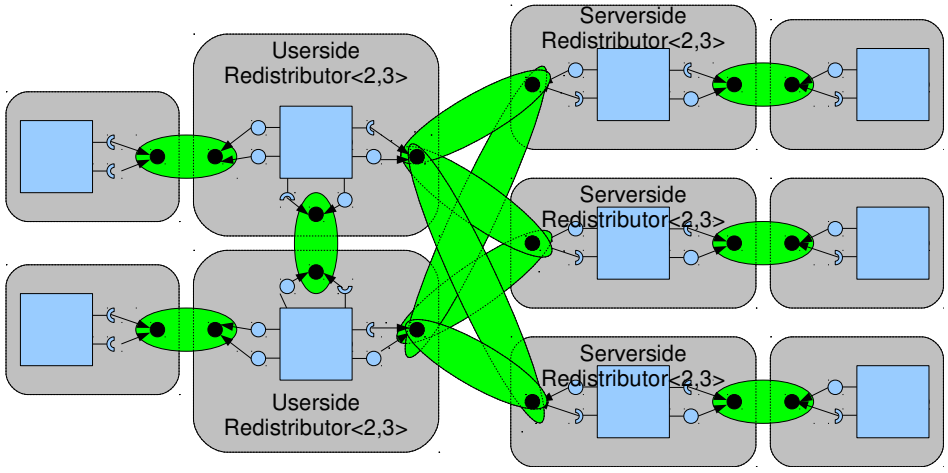

FIGURE 7.25 – Résultat de l'utilisation du générateur `MXNServiceUseProvide` pour le couplage par appel de méthode de deux codes parallèles de degré 2 et 3.

```
component UsersideRedistributor<IntegerM, Integer N> exposes
{
  each(i:[1..M]){
    UseProvide<user={}, provider={UserPartialServiceFacet}> user[i];
  }
  each(i:[1..N]){
    UseProvide<user={ProviderPartialServiceReceptacle}, provider={}> internal[i];
  }
  Collective<member={Receptacle<Barrier>}> barrier;
}
```

FIGURE 7.26 – Définition en HLCM du composant `UsersideRedistributor`.

ment, si elles sont créées dans les processus du composant serveur, c'est du côté serveur que s'effectue la redistribution. Finalement si les instances `userside` sont distribuées avec les processus clients et que les instances `serverside` sont créées du côté du serveur, alors la redistribution a lieu «sur le fil».

7.3.4 Analyse

Expressivité En terme d'expressivité du modèle, cet exemple permet tout d'abord d'exhiber la possibilité qu'il offre de décrire des interactions dont la mise en œuvre s'appuie sur des assemblages relativement complexes comme le sont les interactions par appels parallèles de méthodes. Cet exemple permet cependant aussi d'identifier différentes limitations du modèle.

Un premier aspect qui pose problème dans cette mise en œuvre est du au fait que les mises en œuvres primitives de composant utilisées dans les adaptateurs de connexion et dans le générateur sont liées à l'interface utilisées pour l'interaction. Cet aspect implique que pour décrire des interactions entre composants à l'aide d'un service différent, une grande partie du code devrait être ré-écrit.

On peut cependant noter que cette limitation est principalement due au fait que CCM ne supporte pas le concept de généricité. Il n'est donc pas possible d'y décrire une mise en œuvre primitive de composant qui applique un traitement similaire avec des ports dont les interfaces peuvent varier. L'utilisation d'un modèle sous-jacent qui supporte le concept de généricité serait donc intéressant dans ce cadre.

Il faut cependant noter que dans l'exemple étudié, le service étudié comportait une unique méthode avec un unique paramètre. Cette simplification permettrait de décrire la distribution des données à l'aide d'un unique paramètre générique. Dans le cas de services qui comportent plusieurs méthodes comportant elles-mêmes plusieurs paramètres, la description de la distribution peut poser problème. En effet, au sein de HLCM, une telle interface est considérée comme un élément primitif et il n'est pas possible de s'appuyer sur des informations de grain plus fin.

Une approche qui pourrait être envisagée pour permettre de s'appuyer sur ces informations au sein de HLCM pourrait consister à exporter depuis le modèle sous-jacent des éléments primitifs de grain plus fin. On pourrait par exemple envisager de considérer que les points d'interactions d'un composant ne sont pas les interfaces qu'il fournit ou utilise mais les méthodes au sein de ces interfaces. Le concept d'interface qui regroupe plusieurs méthodes pourrait alors éventuellement être mis en relation avec le concept de *bundle*. On pourrait même envisager d'aller plus loin en exposant chaque paramètre d'une méthode indépendamment. Cet approche pose cependant des questions quant à la sémantique de tels *bundles* et la possibilité de connecter un élément du *bundle* indépendamment des autres.

Une autre approche pourrait consister à proposer des mises en œuvre primitives de composants génériques dont le code est généré en fonction des paramètres. Cette approche correspond notamment à ce qui est fait dans de nombreux modèles d'appel de méthode à distance comme par exemple CORBA avec la génération de souches et de squelettes.

Performances Afin d'évaluer les performances de ces différentes mises en œuvre, elles ont été utilisées au sein d'un code synthétique d'évaluation. Il s'agit d'effectuer 5000 appels à la méthode `apply` dans une boucle au sein du composant client en faisant varier la taille des données. Ces tests on été effectués en utilisant trois processus côté client et quatre côté serveur déployés chacun sur une machine différente.

Ces tests ont été conduits sur la grappe de calcul «Capricorne» de la plateforme expérimentale *Grid'5000*. Cette grappe de calcul est constituée de machines «eServer 326» d'IBM dotées de deux processeurs «Opteron 246» d'AMD. La connexion physique qui a été utilisée entre ces nœuds de calcul est un réseau *Gigabit Ethernet* entièrement commuté. La latence entre ces nœuds est d'environs 50 microsecondes.

Trois mises en œuvres différentes ont été comparées dans cette expérience :

1. la mise en œuvre présentée en 7.3.3 qui s'appuie sur un générateur dédié au cas $M \times N$ avec une redistribution «sur le fil» ;
2. la mise en œuvre présentée en 7.3.2 qui s'appuie uniquement sur les adaptateurs de connexion, sans générateur dédié et qui sérialise donc les appels ; et
3. une mise en œuvre au sein de l'environnement dédié PACO++.

La version de CCM qui a été utilisée s'appuie sur un compilateur développé en interne pour s'appuyer sur la mise en œuvre de CORBA OMNIORB [32, 40] qui présente l'avantage d'offrir des performances intéressantes. C'est cette même mise en œuvre de CORBA qui a été utilisée pour PACO++.

Les résultats obtenus sont présentés sur les figures 7.27 pour ce qui est de la durée des appels et sur la figure 7.28 pour ce qui est de la bande passante utilisée. On peut tout d'abord remarquer que les performances obtenues avec la mise en œuvre basée sur un connecteur HLCM/CCM sont comparables à celles que permet d'obtenir PACO++ voir légèrement meilleures pour les grandes tailles de données. Les performances obtenues avec HLCM sans générateur dédiée qui sérialisent l'appel sur un unique nœud de calcul sont moins bonnes, comme on pouvait s'y attendre.

On se rend compte sur la figure 7.27 que pour les petites tailles de données, cette expérience est dominée par une durée indépendante de la taille des données. Cette durée peut être expliquée en partie par les latences réseau.

Dans le cas d'une redistribution «sur le fil», il n'y a pas de dépendance entre les transmissions de messages, et la latence réseau devrait donc apparaître deux fois (une fois pour

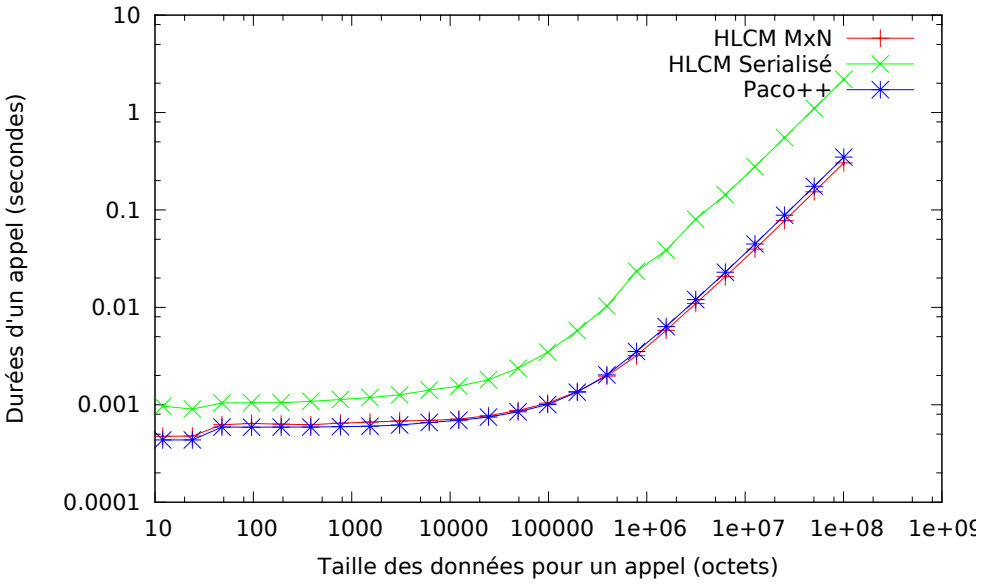

FIGURE 7.27 – Comparaison de la durée d'exécution d'un appel parallèle de méthode $M \times N$ dans le cas où $M = 3$ et $N = 4$. Trois mises en œuvres sont comparées : deux versions qui s'appuient sur HLCM dont l'une sérialise l'appel et l'autre non et une version qui s'appuie sur l'environnement dédié PACO++. L'axe des ordonnées présente la taille du vecteur distribué utilisé comme paramètre de l'appel selon une échelle logarithmique. L'axe des abscisses présente la durée nécessaire à l'exécution de l'appel selon une échelle logarithmique.

l'appel, une fois pour le retour). Dans nos mises en œuvres, les appels sont cependant effectués l'un après l'autre car c'est le comportement de CORBA qui nécessiterait la créations de processus légers(*threads*) pour le contourner. Dans la mesure où chaque instance du côté client envoie des données à deux instances du côté serveur c'est donc quatre fois la latence qui doit être payée, soit 200 microsecondes.

Il faut cependant aussi prendre en compte le temps nécessaire à l'exécution de la barrière qui a lieu après chaque échange réussi. Cette barrière nécessite à un échange de message avec un processus qui centralise les appels pour s'assurer que tous ont atteint la barrière. Dans le cas de la mise en œuvre dans HLCM, cette barrière s'appuie cependant aussi sur CORBA et le processus central libère les deux autres processus l'un après l'autre. Il faut donc compter la durée de deux échanges de message, soit 200 microsecondes. Dans le cas de PACO++, c'est une barrière MPI qui est utilisée et on peux espérer qu'elle soit optimisée pour ne nécessiter que la durée d'un échange de message soit 100 microsecondes. Il faut cependant noter que pour permettre l'utilisation d'une bibliothèque d'ordonnancement de l'envoi des messages, PACO++ exécute une barrière supplémentaire nécessitant donc à nouveau 100 microsecondes.

Les deux mises en œuvres devraient donc nécessiter 400 microsecondes de temps incompressible du aux latences réseau. Aussi bien dans le cas de PACO++ (436 μs) que de HLCM/CCM (474 μs) on voit que le temps minimum d'exécution est plus élevé que ce que les latences réseau seules expliquent. On peut considérer que c'est l'utilisation de CORBA ainsi que l'utilisation d'une mise en œuvre de la barrière non optimale qui expliquent ces sur-coûts.

Dans le cas où les appels sont sérialisés, ce sont six échange de messages qui sont effectués. À nouveau, il faut ajouter à ces chiffres deux échanges de messages supplémentaires pour la barrière. La durée minimale théorique due aux latences réseau est donc de 800 microsecondes et la durée effectivement observée est de 968 μs. Le sur-coût est donc à peu près similaire à ce qui a été constaté dans le cas non sérialisé.

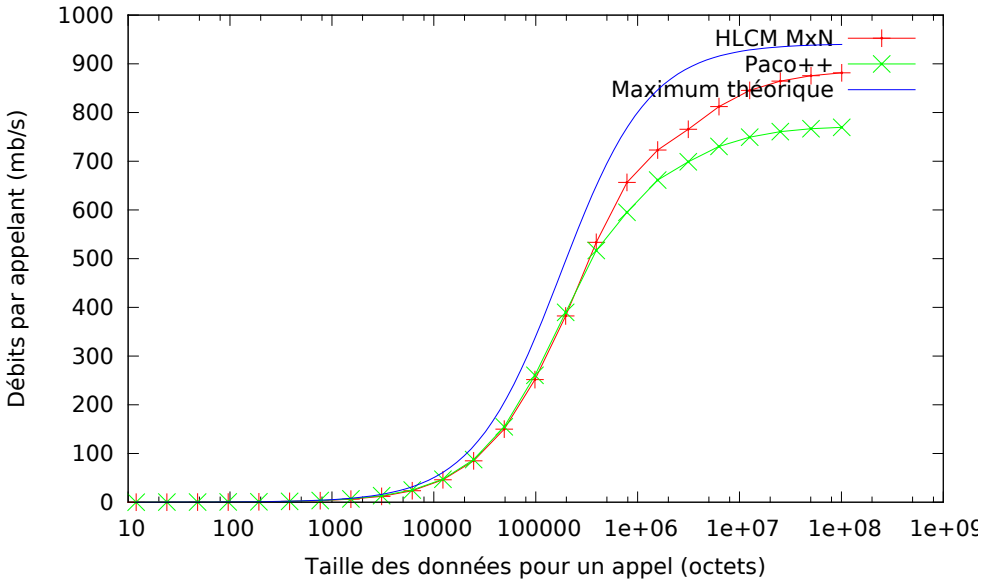

FIGURE 7.28 – Comparaison de la bande passante obtenue sur les appelants lors de l'exécution d'un appel parallèle de méthode $M \times N$ dans le cas où $M = 3$ et $N = 4$. Les résultats obtenus en redistribuant les données sur le fil avec PACO++ et HLCM/CCM sont comparés au maximum théorique. L'axe des ordonnées présente la taille du vecteur distribué utilisé comme paramètre de l'appel selon une échelle logarithmique. L'axe des abscisses présente la bande passante obtenue divisée par trois pour la ramener à chaque processus appelant.

Il faut cependant noter que lorsque la taille des données augmente, la part fixe de la durée d'exécution diminue. Il existe en effet une part de la durée d'exécution qui est directement proportionnelle à la taille des données, ce qui se traduit sur la figure 7.27 adopte une pente constante après la pente nulle du début. Cette durée s'explique principalement par le temps nécessaire au transfert des données, c'est à dire par le débit réseau.

On peut remarquer sur cette figure que l'écart entre la courbe qui correspond au cas sérialisé et au cas classique restent à une distance à peu près constante. Dans la mesure où la figure adopte une échelle logarithmique, cet écart constant sur la figure correspond en fait à des écarts de durée de plus en plus importants. Ce fait est dû à ce que la quantité de transferts effectué est plus important dans le cas sérialisé. En effet, l'ensemble des données sauf celles qui sont sur le nœud appelant y sont regroupé, l'ensemble des données est transmis à l'appelé, puis à nouveau l'ensemble des données sauf celles destinées à ce nœud sont retransmises. À l'inverse avec le générateur dédié au cas $M \times N$, chaque élément du vecteur n'est transmis qu'une fois.

La figure 7.28 permet de mieux analyser ce qui se passe dans le cas où la redistribution est faite «sur le fil». Elle compare les performances obtenues avec PACO++, avec HLCM/CCM et le maximum théorique. Les résultats obtenus sont présentés en terme de débit par appelant, c'est à dire que le débit total est divisé par trois. En effet, les échanges sont fait entre plusieurs nœuds et ce sont les trois accès réseau des appelants qui constituent le goulet d'étranglement. Le maximum théorique est tracé en prenant en compte une durée fixe de 500 microsecondes et un débit de 941 megabits/seconde qui prend en compte les sur-coûts du aux protocoles ETHERNET et TCP.

Contrairement à ce qu'on aurait pu attendre, l'utilisation de CORBA pour les communication entre le code utilisateur et le code de redistribution dans le HLCM/CCM ne le rend pas moins rapide que PACO++ qui s'appuie sur des appels classiques. Un début d'explication est que dans le cas où l'appelant et l'appelé sont situés dans le même processus, OMNIORB permet une optimisation qui évite une recopie inutile de données et supprime donc tout

sur-coût proportionnel à la taille des données.

Il faut cependant noter que PACO++ offre des performances moins bonnes que HLCM/CCM dans le cas de grande tailles de données (au dessus de 400 ko, soit 130 ko par appelant). La version qui s'appuie sur HLCM/CCM atteint 94% du débit maximal théorique avec 881 mb/s alors que la version qui s'appuie sur PACO++ atteint seulement 83% de ce maximal théorique avec 770 mb/s.

Cette différence peut s'expliquer par le fait que la version codée dans HLCM effectue les envois depuis chaque client vers deux serveurs l'un après l'autre alors que PACO++ les effectue de manière simultanée. Le résultat dans cette situation précise est que la version basée sur HLCM n'induit pas de contention alors que PACO++ peut surcharger le lien réseau au niveau des serveurs. C'est la raison pour laquelle PACO++ permet l'utilisation de bibliothèques d'ordonnancement des envois de messages. Aucune bibliothèque de ce type n'a cependant été utilisée au cours de cette expérience.

Interface d'utilisation En terme d'interface d'utilisation, cet exemple démontre les possibilités de HLCM. Le comportement «boîte noire» des composants est assuré en offrant une même interface quelle que soit la mise en œuvre utilisée, séquentielle ou parallèle. La conservation de ce comportement ne se fait cependant pas au détriment des performances puisque dans le cas $M \times N$, il reste possible de s'appuyer sur un générateur spécifique.

Une limitation de l'approche adoptée est cependant qu'il est nécessaire de connaître les mises en œuvres des composants utilisés pour déterminer quel générateur doit être proposé. Un utilisateur qui s'appuierait sur des composants dont il ne sait pas qu'ils sont parallèle ne pourrait par exemple pas aisément déterminer que le développement d'un générateur gérant le cas $M \times N$ permettrait d'améliorer les performances. Cette limitation est cependant intrinsèquement lié au fait de ne pas exposer les détails de la mise en œuvre dans l'interface du composant.

Il faut cependant noter que la découverte des informations nécessaires pour déterminer les mises en œuvres améliorés qui pourraient être proposées ne nécessite l'étude que de l'interface des mises en œuvre et non pas de leur comportement profond. On pourrait aussi imaginer le développement d'un outil qui analyse un assemblage transformé pour en extraire l'ensemble des adaptateurs de connexions utilisés qui pourraient éventuellement être remplacés par des générateurs dédiés.

En ce qui concerne la séparation des préoccupations, HLCM permet comme dans le cas du partage de mémoire de séparer clairement le code métier du code plus technique.

7.4 DISCOGRID

Un troisième volet de l'évaluation de HLCM a consisté à l'utiliser dans le cadre du modèle DISCOGRID [42, 41]. Il s'agit d'un modèle inspiré de MPI dont l'objectif est de mieux prendre en compte l'aspect hiérarchique des ressources au sein des grilles de calcul. Les processus n'y sont donc plus identifiés par un simple identifiant scalaire mais par un vecteur qui identifie une feuille d'une structure en arbre comme présenté sur le schéma de la figure 7.29. Les données sont découpées de manière similaire et chaque nœud dans l'arbre des processus est associé à une partie des données.

Cette approche permet d'exprimer des opérations de communication collective en terme de communication point à point entre les nœuds internes de l'arbre. C'est par exemple le cas pour l'échange des données aux frontières entre les matrice A_0 et A_1 de la figure 7.29. Cet échange peut être exprimé sous la forme d'un échange entre les nœuds 0.0 et 0.1 de l'arbre des processus. Il n'est pas nécessaire au sein de l'application d'effectuer de traitement particulier pour prendre en compte le fait qu'il s'agit d'une opération de communication de type MxN nécessitant une redistribution des données.

Comme dans MPI, des choix doivent être fait lors du déploiement des applications. Cependant puisque le modèle est hiérarchique, il ne s'agit pas seulement de choisir le nombre de processus à utiliser mais la structure de l'arbre. Pour chaque nœud de l'arbre, il faut

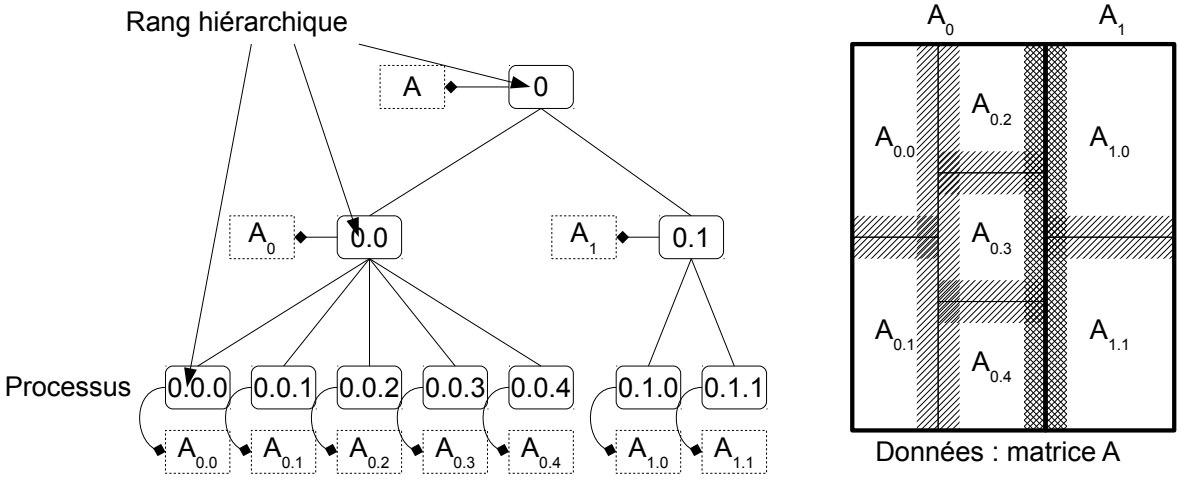

FIGURE 7.29 – Présentation du modèle DISCOGRID. Les processus sont regroupés selon la hiérarchie des ressources et les données découpées de manière similaire leurs sont attribuées. Ce découpage permet d'identifier la part des données attribuée à chaque machine, chaque grappe ou à la grille complète.

déterminer s'il s'agit d'une feuille —et dans ce cas nœud est associé à un processus— ou s'il représente un groupe (logique) et alors déterminer le nombre de nœuds fils.

Des travaux [21] ont montré que dans le cadre de véritables applications, l'approche naïve qui consisterait à utiliser l'ensemble des nœuds de calcul disponible dans une grille et à les regrouper en suivant la hiérarchie n'est pas toujours optimale. Pour l'application de modélisation électromagnétique (*Computational ElectroMagnetics* – CEM) sur laquelle ils s'appuient, ces travaux proposent plusieurs heuristiques qui prennent en compte les ressources et le maillage appliqué aux données. Ces heuristiques permettent de déterminer le nombre de processus et leur regroupement hiérarchique.

Cet exemple présente un cadre intéressant pour évaluer dans quelle mesure HLCM et $HLCM_i$ permettent la mise en œuvre d'heuristiques de déploiement existantes en tant qu'algorithmes de choix de mise en œuvre. La mise en œuvre d'une de ces heuristiques au sein de $HLCM_i$ a été étudiée en collaboration avec Cristian Klein qui l'avait proposée au cours de ses travaux de master. Avant de s'intéresser à la mise en œuvre de l'algorithme de choix de mise en œuvre, il faut toutefois commencer par définir le modèle au sein duquel décrire les applications elles-même.

7.4.1 Modélisation

Deux options peuvent être envisagées pour la modélisation des applications basées sur DISCOGRID dans HLCM :

- une première option consiste à mettre en œuvre les éléments de DISCOGRID dans une spécialisation de HLCM existante à la manière de ce qui a été fait pour le partage de données ou les appels parallèles de méthodes dans HLCM/CCM ;
- une seconde option consiste à définir une nouvelle spécialisation de HLCM dédiée.

En ce qui concerne la mise en œuvre d'heuristiques pour le choix de mise en œuvre des composants dans $HLCM_i$, ces deux approches sont équivalentes. La seconde option a toutefois l'avantage de ne nécessiter aucune modification des applications et c'est donc celle qui a été adoptée.

Ces travaux s'appuient donc sur une mise en œuvre existante de DISCOGRID appelée *Data HIerachical COmponents* (DHICO). Cette mise en œuvre se présente sous la forme d'une bibliothèque qui offre les différentes primitives du modèle DISCOGRID en s'appuyant sur des bibliothèques de communication existantes. Plusieurs configurations sont possibles,

```
component DhicoSkeleton<component C, DhicoData d> exposes {}
```

Figure 7.30 – Squelette Dhico dont la mise en œuvre est primitive. Il prend en paramètre un composant C qui correspond à l'application à exécuter et des données `d` à lui passer. Il n'expose aucune connexion ouverte.

mais la seule configuration que nous avons considérée consiste à utiliser MPI pour les communications au sein des grappes de calcul et des *sockets* TCP pour les communications entre les grappes.

Comme pour MPI, l'exécution des applications qui s'appuient sur cette bibliothèque se fait à l'aide d'un outil dédié responsable du lancement des processus sur les différents nœuds de calcul. Cet outil est basé sur l'environnement de déploiement Adage.

Pour déployer l'application, cet outil prend en entrée une description des ressources matérielles, une description de l'application et une description du placement de l'application sur les ressources. La description de l'application est constituée de la spécification de l'exécutable concerné, mais aussi des données devant être distribuées pour permettre son exécution. Le placement de l'application est décrit en spécifiant simplement le nombre de processus devant être déployé sur chaque grappe de calcul puisqu'il est considéré que ces dernières sont homogènes.

Les paragraphes qui suivent présentent les choix qui ont été effectués pour la définition d'une spécialisation de HLCM basée sur Dhico. Le but n'est pas de présenter dans le détail la spécialisation mais de souligner ses aspects particuliers et les raisons qui les justifient ainsi que les aspects qui impactent la mise en œuvre de l'heuristique de choix des ressources et donc de la structure de l'application à déployer.

Squelette Dhico primitif Afin de définir une spécialisation de HLCM basée sur Dhico, il est nécessaire d'identifier les éléments primitifs qu'il fournit ainsi que les éléments primitifs qui peuvent y être décrits. Une première analyse pourrait tendre à faire modéliser les applications écrites pour Dhico comme des composants et la bibliothèque elle-même sous la forme d'un connecteur dont la mise en œuvre est primitive.

Il faut cependant noter que l'utilisation de Dhico pour les communications au sein d'une application impose des contraintes fortes sur l'architecture de celle-ci. L'outil de déploiement utilisé pour exécuter les applications n'est par exemple pas prévu pour traiter différemment un processus donné au sein de l'application. Cette contrainte empêche par exemple de décrire une application où un processus donné participe à deux interactions utilisant Dhico avec des groupes de processus complètement indépendants par ailleurs.

On peut donc noter que Dhico ne décrit pas qu'une interaction entre des éléments arbitraires, mais plutôt une structure d'application comportant notamment un type spécifique d'interaction. Ces éléments plaident en faveur de la modélisation de Dhico non pas comme un connecteur mais comme un squelette algorithmique à la manière par exemple des composants parallèles présentés dans les sections précédentes. C'est le composant `DhicoSkeleton` présenté sur la figure 7.30 qui est utilisé dans ce but. Ce composant accepte comme paramètre générique `C`, un autre composant et `d`, la description des données traitées. Le composant `C` doit exposer une connexion ouverte qui correspond aux interactions proposées par DiscoGrid.

La mise en œuvre de ce squelette est primitive puisqu'elle est gérée directement par le modèle sous-jacent Dhico. Cette mise en œuvre primitive décrite de manière informelle sur la figure 7.31 comporte des paramètres libres dont les valeurs sont transmises à l'outil de déploiement Dhico. Les deux premier paramètres libres de la mise en œuvre correspondent aux paramètres du composant lui-même : le composant à exécuter et les données à traiter. Ces informations ne sont cependant pas suffisantes pour décrire une exécution de l'application ; il faut en plus spécifier comment la hiérarchie est organisée. Cette hiérarchie est passée à l'outil de déploiement sous la forme de la spécification du nombre d'instances

```
DhicoImpl<component CMP, DhicoData data, Vector<Integer> distrib>
    implements DhicoSkeleton<CMP, data>;
```

FIGURE 7.31 – Description informelle de la mise en œuvre primitive du composant `DhicoSkeleton`. Cette mise en œuvre possède trois paramètres libres *CMP*, *data* et *distrib* dont les valeurs sont transmises à l'outil de déploiement.

devant être créées sur chaque grappe. Cette information possède deux facettes :
- le nombre de groupes d'instances (déployées chacun sur une grappe) et le nombre d'instance dans chaque groupe ; cette facette modifie l'application elle-même ;
- l'association de chacun de ces groupes d'instances à une grappes donnée ; cette facette ne change en rien l'application elle-même mais seulement le plan de déploiement.

La première facette est donc décrite par un paramètre libre de la mise en œuvre qui prend la forme d'un vecteur d'entiers. La seconde est absente du modèle de description de l'application et est restreinte au plan de déploiement.

Il n'existe pas au sein de cette spécialisation de possibilité de décrire de générateurs et il n'en existe pas de primitif. Les interactions entre composants sont donc restreintes à celles permises par le squelette `DhicoSkeleton`. Les composites qui peuvent être décrits se restreignent donc à la juxtaposition d'instances indépendantes du squelette.

Ce modèle permet l'utilisation de l'infrastructure de HLCM_i pour mettre en œuvre de l'heuristique de placement. Il est cependant suffisamment particulier pour que la question de l'applicabilité des résultats ainsi obtenus à des modèles plus généralistes puisse se poser. Ces considérations ont mené à la définition d'un second modèle qui vise à se rapprocher plus des modèles de composant classique.

Connexions *socket* et squelette MPI Décrire les applications utilisant DHICO d'une manière plus proche des modèles de composants classiques nécessite de les considérer comme un ensemble d'instances en interaction. Un second modèle a donc été créé dans le but de décrire une spécialisation de HLCM au sein de laquelle un squelette similaire au squelette DHICO précédemment présenté puisse aussi être décrit mais avec une mise en œuvre composite plutôt que primitive. Pour atteindre ce but, il faut renoncer à utiliser l'outil de déploiement de DHICO qui impose des contraintes trop forte sur la structure de l'application. Il faut donc étudier son comportement afin de permettre de le reproduire à l'aide d'une spécialisation de HLCM plus généraliste.

Cet outil de déploiement s'appuie en fait principalement sur l'outil de déploiement MPI qui est utilisé pour déployer l'application sur chaque grappe de calcul. Il passe cependant aux applications ainsi déployées des paramètres additionnels afin qu'elles puissent interagirir pour mettre en œuvre le modèle DISCOGRID. Un paramètre indique à chaque groupe de processus la partie de l'identifiant hiérarchique commune à tous les éléments du groupe. Un autre paramètre indique l'adresse réseau à laquelle peut être contacté l'un des processus de l'application. Ce processus est utilisé comme point de rendez-vous pour permettre des connexions directes à l'aide de *sockets* TCP avec n'importe quel processus étant donné son identifiant hiérarchique.

On peut donc identifier au sein de ces applications deux moyens de communication distincts :

MPI est utilisé pour des interactions au sein de groupes de processus qui sont tous déployés sur la même grappe de calcul ; et

des *sockets* TCP sont utilisées pour des interactions au sein d'un groupe de processus qui peut être déployé sur plusieurs grappes de calcul.

L'interaction au travers de *sockets* fait intervenir un groupe de processus. Parmi ceux-ci, un processus a le rôle spécifique de point de rendez-vous, il s'agit cependant d'un aspect de la mise en œuvre qui est masquée lors de l'utilisation. Chaque processus peut participer à

```
connector SocketConn<role member>;
```

Figure 7.32 – Définition du connecteur `SocketConn` utilisé pour les interactions au travers de *sockets* dans le second modèle. Il possède un unique rôle `member`.

```
component MpiSkeleton<component C, connector EXP> exposes
{
    EXP exposed;
}
```

Figure 7.33 – Squelette MPI dont la mise en œuvre est primitive. Il prend en paramètre un composant C qui correspond à l'application à exécuter et un connecteur EXP qui spécifie le type de la connexion ouverte qu'il expose.

un nombre quelconque d'interactions de ce type sans qu'il soit nécessaire que tous les processus participants aux même interactions. Ces aspects sont modélisés par le connecteur `SocketConn` présenté sur la figure 7.32 qui possède un unique rôle `member`. Ce connecteur est mis en œuvre par un générateur primitif.

Pour les interactions au travers de MPI, on peut noter qu'elles imposent à peu près les mêmes contraintes que celles qui avaient été identifiées pour Dhico dans le premier modèle. Dans ce cas aussi, un outil de déploiement dédié est utilisé qui rend notamment impossible pour un processus de faire partie de deux groupes de communication distincts. Ces considérations ont conduit à la modélisation de MPI comme un squelette algorithmique dans ce second modèle.

Comme pour le composant `DhicoSkeleton` du premier modèle, le composant `MpiSkeleon` présenté sur la figure 7.33 prend en paramètre un composant `C` qui correspond au code à exécuter en parallèle. Sa mise en œuvre est primitive et possède un premier paramètre libre qui correspond au code à exécuter et un second qui décrit la réplication. Contrairement au squelette `DhicoSkeleton`, le degré de réplication n'est pas décrit par un vecteur d'entiers mais par un simple entier qui spécifie le nombre d'instances à créer.

À la différence du cas Dhico, on peut noter qu'il est possible de passer au squelette des informations qui seront utilisées par le composant interne. Il est ainsi possible de décrire des connexions *socket* auxquelles devra participer le composant interne. Il est aussi possible de passer en paramètres des valeurs comme par exemple une partie de l'identifiant hiérarchique. Dans les deux cas, l'outil de déploiement MPI transmet les paramètres à l'ensemble des processus sur une grappe, ce qui signifie en terme de composant que l'ensemble des instances dans le squelettes seront traitées de manière similaire.

En ce qui concerne le passage de valeurs de données aux instances internes du squelette, on peut les modéliser sous la forme de paramètres génériques directement spécifiés pour le type de composant utilisé comme paramètre générique du squelette. En effet, il s'agit de paramètres qui sont fixés lors du déploiement et qui influent sur le comportement des composants. Cet aspect est illustré sur la figure 7.34.

En ce qui concerne les connexions, le fait qu'un squelette soit modélisé par un composant impose que l'ensemble des interactions auxquelles participent les instances internes apparaissent dans l'interface du squelette. C'est donc le squelette entier qui expose une connexion ouverte destinée à interagir au travers de *sockets* et cette connexion se traduit dans sa mise en œuvre par une participation de l'ensemble des instances qu'il contient à la

```
MpiSkeleton<Component<ID="1.1">, SomeConnector>
```

Figure 7.34 – Passage de paramètre au type de composant lui-même utilisé comme paramètre du squelette MPI.

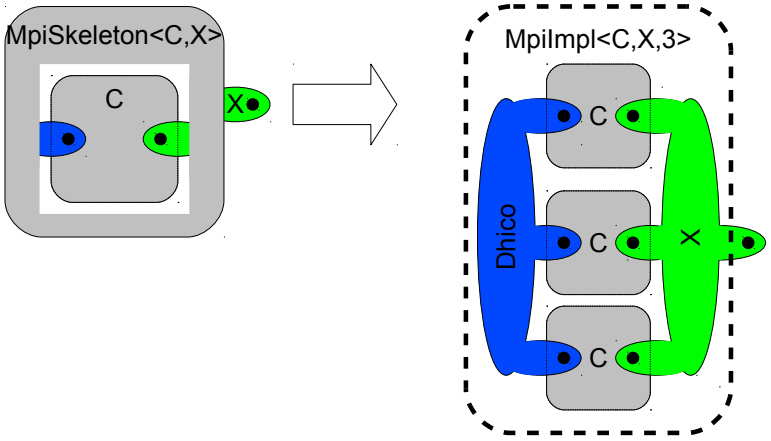

FIGURE 7.35 – Mise en œuvre primitive du squelette modélisé par le composant `MpiSkeleton`. La mise en œuvre expose une connexion ouverte qui rassemble les connexions exposées par les instances du composant répliqué.

```
component DhicoSkeleton<component C> exposes {}

composite DhicoSkeletonImpl<component C, Integer N>
implements DhicoSkeleton<C>
{
  each(i:[1..N]){
    MpiSkeleton<C, SocketConn> inst[i];
  }
  merge(each(i:[1..N]){inst[i].exposed});
}
```

FIGURE 7.36 – Description en HLCM du squelette `DhicoSkeleton` et de sa mise en œuvre composite `DhicoSkeletonImpl`.

connexion comme présenté sur le schéma de la figure 7.35.

Il faut toutefois noter qu'il a été spécifié que les processus pouvaient participer à plusieurs interactions distinctes au travers de *sockets*. Le fait qu'ils soient exécutés par l'outil de déploiement de MPI n'impose pas de contrainte supplémentaire puisque le nombre de paramètres pouvant être transmis aux processus n'est en rien restreint. À l'inverse, dans HLCM, il n'est pas possible pour un composant d'exposer un nombre inconnu de connexions ouvertes.

La modélisation de cet aspect peut toutefois s'appuyer sur le concept de *bundle* qui permet de regrouper plusieurs connexions ouvertes. En utilisant un paramètre générique du composant pour définir le type de la connexion ouverte exposée, il est possible d'y utiliser une connexion comportant un *bundle* qui regroupe un nombre arbitraire de connexions. C'est le rôle du paramètre `EXP` présenté sur la figure 7.33.

Le squelette DHICO peut être mis en œuvre sous la forme d'un composite s'appuyant sur le composant générique `MpiSkeleton` et sur le connecteur `SocketConn` comme présenté sur la figure 7.36. Cette mise en œuvre contient un nombre d'instances du squelette MPI spécifiée par le paramètre libre `N`. Ces instances exposent des connexions ouvertes `exposed` de type `SocketConn` qui sont toutes fusionnées.

7.4.2 Mise en œuvre au sein de HLCM$_i$

En plus de la définition de modèles adaptés, la mise en œuvre des deux spécialisations basées sur DHICO au sein de HLCM$_i$ s'appuie sur un composant qui met en œuvre l'algorithme de choix et un composant qui génère les fichiers nécessaires au déploiement. Ces mises en œuvre prennent en entrée des fichiers HLCM et génèrent en sortie les fichiers devant être passés à l'outil de déploiement.

On peut noter que dans l'état actuel, certains assemblages décrits à l'aide du second modèle présenté donnent lieu à des configurations que l'outil de déploiement n'est pas encore capable de gérer, empêchant ainsi le déploiement de l'application. Ces assemblages correspondent à des assemblages qui n'auraient pas pu être décrits à l'aide du premier modèle. Cette limitation n'impacte cependant pas l'évaluation du code de choix de mise en œuvre. En effet, l'étude de l'assemblage généré suffit à s'assurer que ce sont bien les bons choix qui ont été effectués

Mise en œuvre de l'algorithme de choix L'heuristique mise en œuvre pour guider le déploiement de l'application sur des ressources de type grille —ou fédération de grappes— est celle proposée par l'algorithme appelé *Greedy* (c'est un algorithme glouton) dans les travaux originaux [21]. Cet algorithme s'appuie sur un modèle de performances des applications basées sur DHICO. Il effectue une estimation du temps nécessaire à l'exécution de l'application pour un certain nombre de déploiements. Il commence avec un ensemble de ressources vide, puis cherche à ajouter les grappes une par une. Le nombre de grappes utilisées est ainsi augmenté jusqu'à ce que le modèle de performance ne prédise plus d'amélioration du temps d'exécution.

Cet algorithme de choix est donc spécifique à DHICO, c'est à dire qu'il est associé au composant qui correspond au squelette DHICO dont les paramètres sont similaires dans les deux modèles décrits. Il s'appuie de plus sur des informations spécifiques à l'application qui utilise DHICO : le type et la fréquence des interactions et la quantité de données transmises. Ces informations sont associées au composant pris en paramètre par le squelette.

Du point de vue de la mise en œuvre de l'algorithme de choix, il s'agit donc principalement de reprendre le composant de choix naïf qui avait été présenté en 6.4.2 et de mettre un œuvre un cas particulier pour un type de composant spécifique. Il semblerait intéressant de permettre de définir un tel algorithme associé à un type de composant sous la forme de meta-information, à l'aide d'un langage de script. Cette solution éviterait de devoir proposer une nouvelle mise en œuvre du compilateur pour chaque nouveau composant auquel est associée une heuristique dédiée. Une solution devrait aussi être trouvée par la spécification des paramètres nécessaires à ces scripts et qui peuvent être liés aux paramètres du composant. C'est par exemple le cas des paramètres de l'heuristique présentée ici qui sont liés à l'application déployée. Dans le cadre de nos expérimentations, cette heuristique a cependant été directement codée au sein du composant de choix de mise en œuvre de HLCM$_i$. Ces paramètres sont simplement spécifiés en dur en fonction de l'application trouvée en paramètre.

L'algorithme correspondant à l'heuristique elle-même est relativement simple. Sa mise en œuvre nécessite moins de 250 lignes de code JAVA. On peut cependant noter que cet algorithme s'appuie fortement sur le concept de latence entre grappes, or le modèle de ressource d'ADAGE utilisé dans HLCM$_i$ expose des informations sur le concept plus général de latence entre nœuds. Du code qui calcule les valeur utilisées à partir de celles disponibles dans le modèle et qui sauvegarde ces informations fréquemment utilisées a donc été développé. Il compte une cinquantaine le ligne de code JAVA. Une dernière partie du code du composant de choix concerne l'accès au modèle et au plan de déploiement pour y inscrire les informations calculées. Cette partie du composant comporte aussi une cinquantaine de lignes de code.

Génération des fichiers finaux La génération des fichiers nécessaires à l'exécution de l'application est effectuée par un composant dédié qui remplace le composant `CCM Dumper` qui

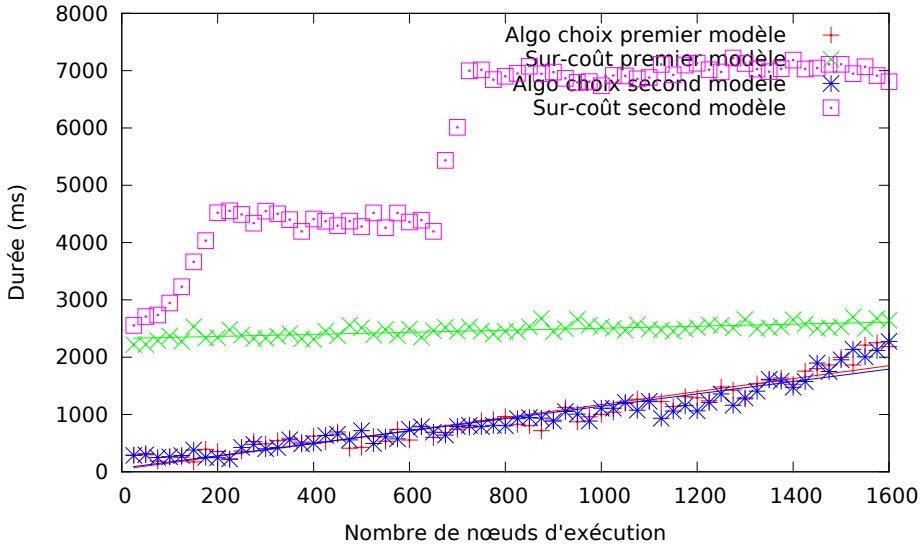

FIGURE 7.37 – Durée d'exécution de l'opération de transformation pour l'application de CEM décrite pour les deux spécialisations de HLCM présentées dans ce chapitre. Dans chacun de ces cas, la durée d'exécution de l'algorithme de choix lui même est dissociée des sur-coûts induits par l'utilisation de HLCM.

était décrit dans la figure 6.8. Ce composant accède au modèle de l'application, au modèle des ressources et au plan de déploiement. Il génère des fichiers nécessaires à l'exécution de l'application.

Ces fichiers dans un format dédié sont destinés à l'outil de déploiement de DHICO. Ils comportent la description d'un ensemble de déploiement d'applications MPI avec les paramètres qui doivent leur être passé et les données qui doivent leur être fournies.

7.4.3 Analyse

Les deux spécialisations de HLCM qui ont été définies dans cette section —et qui ont été implémentées— démontrent qu'il est possible d'adapter HLCM à des modèles relativement éloignés des modèles de composant classiques. Elles montrent aussi que l'utilisation de HLCM et en particulier de $HLCM_i$ offrent un cadre intéressant pour la mise en œuvre d'heuristiques d'adaptation des applications aux ressources d'exécution.

Du point de vue des performances, on peut noter que l'outil de déploiement de DHICO est conservé et que $HLCM_i$ n'est utilisé que pour générer sa configuration. L'utilisation du même algorithme pour le choix des paramètres qui lui sont transmis que dans le cas de la mise en œuvre qui ne s'appuyait pas sur HLCM assure que les performances à l'exécution ne soient pas impactées du tout par son utilisation.

Il faut donc chercher les éventuels sur-coûts dans la phase de transformation elle-même. Pour les évaluer, l'application de CEM a été décrite à l'aide des deux modèles présentés dans ce chapitre. La durée nécessaire à l'exécution de $HLCM_i$ sur cette application en fonction de la taille du nombre de nœuds d'exécution disponibles est présentée sur la figure 7.37.

On peut tout d'abord noter sur la figure 7.37 que le coût d'exécution de l'algorithme de choix lui même est similaire pour les deux modèles. Cet aspect est normal puisque c'est exactement le même algorithme qui est utilisé. Ce temps est globalement proportionnel au nombre de nœuds d'exécution disponibles.

On peut aussi noter que le temps de transformation est dominé par les sur-coûts dus à HLCM. Pour les deux modèles ces sur-coûts se caractérisent par un temps incompressible d'environs 2,5 secondes qui correspond à l'initialisation du système et à la lecture

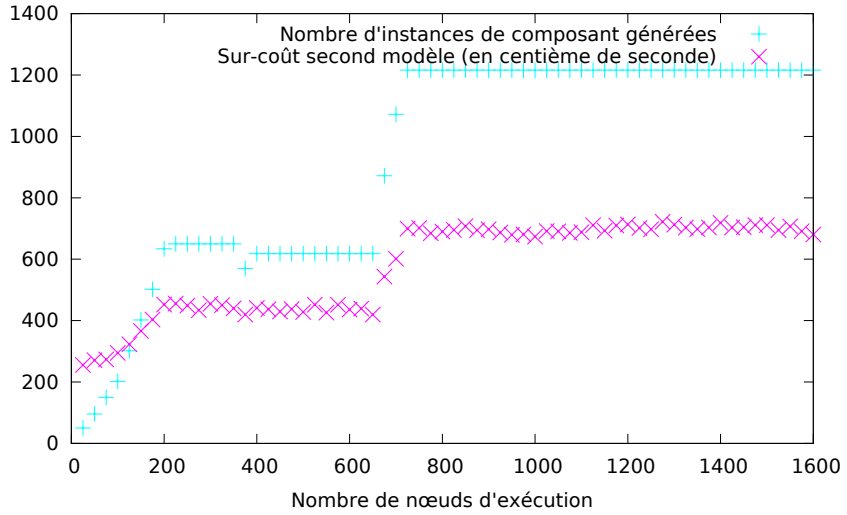

FIGURE 7.38 – Nombre d'instances de composant générées dans le cas du second modèle en fonction du nombre de nœuds d'exécution disponibles.

des fichiers sources. Cependant dans le cas du premier modèle, ce temps est à peu près indépendant du nombre de nœuds d'exécution disponibles alors que dans le cas du second modèle on observe un comportement plus compliqué.

Ce comportement peut s'expliquer par le fait que pour le premier modèle, le nombre d'instances primitives générées reste toujours exactement égal à un. C'est en effet le squelette DHICO qui représente l'application complète qui constitue cette instance primitive. Dans le cas du second modèle cependant, le nombre d'instances générées dépend des choix effectués puisque le grain de modélisation est plus fin comme on peut le voir sur la courbe de la figure 7.38. La figure 7.39 présente l'évolution des sur-coûts dus à HLCM dans ce cas en fonction du nombre d'instances dans l'assemblage final. On voit que ce coût est à peu près linéairement corrélé au nombre d'instances créées.

Ces sur-coûts ne peuvent pas être ignorés puisqu'ils peuvent atteindre l'ordre de grandeur d'une dizaine de secondes pour quelques milliers de composants. Il faut cependant les mettre en regard de la durée nécessaire au déploiement d'une telle application sur des ressources distribuées telles que celles d'une grille de calcul.

Une première analyse des raisons à l'origine de ces coûts a été toutefois été effectuée. Il sont principalement dus à l'utilisation exclusive de liste pour regrouper des collections d'objets dans le code généré par l'infrastructure EMF. Il serait intéressant d'évaluer la possibilité de s'appuyer sur des structures plus optimisées telles que par exemple des tables de hachage dans le cas de collections souvent accédé à l'aide de la même clef. C'est par exemple le cas pour la recherche d'un composant étant donné son nom qui nécessite d'itérer entièrement sur la collection dans le cas d'une liste.

7.5 Conclusion

Ce chapitre a présenté une évaluation du modèle HLCM et en particulier de sa spécialisation HLCM/CCM. Il a aussi proposé une évaluation de sa mise en œuvre au sein de $HLCM_i$. Cette évaluation s'est appuyée sur la mise en œuvre de deux interactions spécifiques au sein de HLCM/CCM : le partage de mémoire et l'appel parallèle de méthode. Elle a aussi évalué une spécialisation spécialement conçue pour la bibliothèque DHICO et pour laquelle une heuristique de choix plus avancée que les algorithmes naïfs autrement utilisés a pu être mis en œuvre.

Cette évaluation a permis de montrer que HLCM propose un modèle intéressant pour le

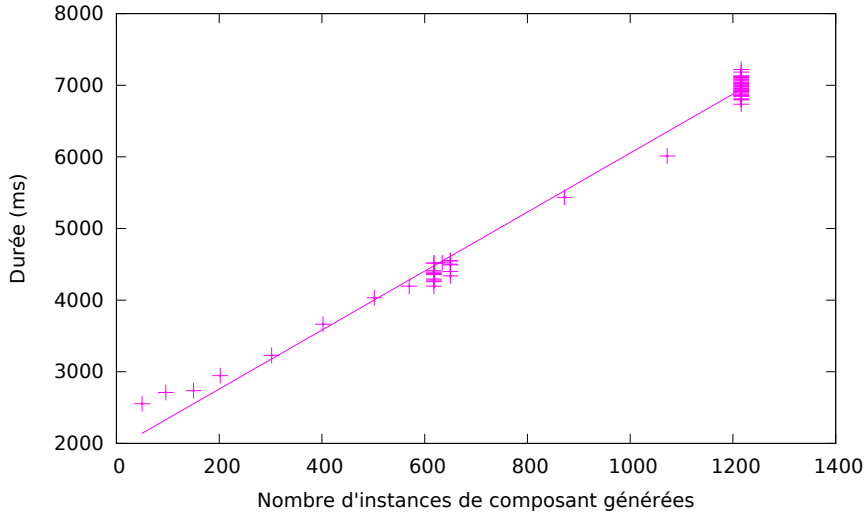

FIGURE 7.39 – Sur-coûts induits par l'utilisation de HLCM dans le cas du second modèle en fonction du nombre d'instances de composant générées.

développement d'application de calcul scientifique à haute performance. Il offre une expressivité qui permet la mise en œuvre au sein des modèles d'une grande gamme de squelettes algorithmiques et d'interactions. Il permet d'obtenir des performances satisfaisante en permettant notamment l'adaptation des applications aux ressources d'exécution sur lesquelles elles sont déployées. Il offre enfin une interface d'utilisation qui si elle comporte quelques concepts nouveaux qui peuvent être déroutants assurent le comportement «boîte noire» des codes et une bonne séparation des préoccupations entre code technique et code métier.

L'évaluation a aussi permis de mettre en évidence certaines limitations de HLCM. Ces limitations sont principalement liées aux modèles sous-jacent utilisés qui n'ont pas été spécifiquement conçus pour une utilisation avec HLCM. Ils sont donc limités par l'absence en leur sein de certains concepts tels que les connecteurs ou la généricité. Leur modélisation au sein de HLCM a aussi été effectuée à un grain qui pourrait être affiné pour permettre d'y manipuler indépendamment jusqu'à chaque paramètre des méthodes par exemple.

D'autres limitations sont dues à la mise en œuvre $HLCM_i$, c'est par exemple le cas des sur-coûts impliqués lors du déploiement. Ces sur-coûts sont liés d'abord à l'utilisation du langage JAVA et ensuite à l'utilisation de code généré par l'infrastructure de modélisation d'EMF, qui ne génère pas encore toujours du code très optimisé. Il faut cependant noter que ces outils ont permis de fortement réduire le temps nécessaire à la mise en œuvre de $HLCM_i$, qu'ils restent raisonnables dans le cadre d'une transformation unique appliquée lors du déploiement et qu'ils peuvent être optimisés.

CHAPITRE 8

Conclusion et perspectives

Conclusion

Contexte d'étude

À la recherche d'une compréhension toujours plus complète du monde, les scientifiques conçoivent des modèles toujours plus précis des phénomènes qui nous entourent. La validation de la cohérence de ces modèles et leur utilisation à des fins de prédiction a donné naissance à un domaine à part entière de l'informatique : le calcul scientifique. Les applications de simulation numérique qui forment ce domaine posent de par leurs spécificités des défis particuliers.

Afin de prendre en compte les interactions entre les différents aspects des objets modélisés, les applications de simulation s'appuient sur de multiples modèles qui prennent chacun en compte un aspect différent. Ces modèles correspondent souvent à différents domaines de la science et sont donc développés par des équipes différentes. Des codes patrimoniaux qui gèrent déjà certains de ces modèles sont souvent réutilisés dans ce cadre. L'architecture des applications est donc complexe et si elle n'est pas bien maîtrisée, la composition des différentes parties qui la composent peut poser problème.

L'ampleur des données traitées au sein de ces applications et la complexité croissante des modèles utilisés nécessitent aussi des quantités de calcul toujours plus importantes. Afin de fournir la puissance de calcul nécessaire, la réponse des concepteurs de calculateurs consiste à agréger la puissance offerte par de multiple cœurs de calcul. Ce regroupement prend plusieurs formes comme les processeurs multi-cœurs, les machines multi-processeurs, les grappes de calcul, les super-calculateurs et les grilles de calcul. Ces architectures matérielles complexes et qui évoluent plus vite que les codes qui y sont exécutés rendent complexe l'optimisation des applications.

Devant la complexité conjointe des applications et des architectures matérielles sur lesquelles elle sont exécutées, il est nécessaire de proposer des modèles de programmation adaptés. Un tel modèle devrait faciliter la construction d'applications à l'aide d'éléments éparses développés indépendamment. Il devrait permettre aux applications qui l'utilisent de s'exécuter efficacement sur un ensemble d'architectures matérielles important parmi lesquelles certaines n'existaient éventuellement même pas lors de la conception du code. Il devrait offrir aux scientifiques qui développent les applications et qui ne sont pas experts dans le fonctionnement interne des machines des abstractions qui leur évitent de se préoccuper des aspects les plus bas niveau.

Les modèles à base de composant logiciels offrent une base intéressante pour la conception d'un tel modèle de programmation. Ils proposent de concevoir les applications sous la forme d'assemblage de composants indépendants, facilitant ainsi la réutilisation de codes existant et développés par des équipes différentes. En abstrayant certains aspects techniques comme la gestion de la distribution ou de la sécurité, ils proposent un environnement de programmation qui permet aux scientifiques de se concentrer sur leur domaine de compétence. Il existe des travaux qui proposent d'intégrer au sein de ces modèles des

abstractions qui correspondent aux paradigmes de programmation pour le parallélisme et la distribution pour permettre aux codes de s'adapter aux évolutions du matériel.

Au cours de notre analyse, nous avons relevé un certain nombre de problèmes liés à cette approche. En se limitant à la gestion d'un nombre limité de paradigmes au sein de chaque modèle, elle pousse à la multiplication des modèles, chacun gérant un ensemble de paradigmes différent. Cette multiplication des modèles pose problème quand il s'agit de réutiliser des codes existant. Elle pose le risque que l'ensemble des paradigmes nécessaires à une application donnée ne soient pas présents au sein d'un même modèle. Finalement, avec cette approche la probabilité de trouver un modèle supportant des paradigmes qui correspondent à des classes restreintes d'applications est très faible.

Contribution

Notre étude a permis d'effectuer une classification des concepts introduits au sein des divers modèles de composants pour supporter les paradigmes de programmation parallèle. Il en existe deux types :

1. les concepts qui proposent l'abstraction d'interactions entre les composants qui forment l'application ; et
2. ceux qui proposent l'abstraction d'architectures de composition connues dans lesquelles seuls quelques éléments doivent être spécifiés.

Pour ces deux types de concepts, l'introduction d'une abstraction pour les réifier permet d'en proposer diverses mises en œuvres et de choisir la plus adaptée lorsque les ressources d'exécution sont connues.

Nous avons aussi noté que ces catégories de concepts correspondent à des éléments qui ont déjà été identifiés. Le concept de connecteur modélise les interactions entre composant. Les squelettes algorithmiques décrivent des architectures de composition connues dans lesquelles seuls quelques éléments doivent être spécifiés.

La thèse que nous avons soutenue dans ce manuscrit est qu'*il est possible de définir un modèle de composants dans lequel de nouveaux opérateurs de composition peuvent être introduits* et qu'*ils peuvent être utilisés pour supporter les divers paradigmes de programmation parallèle*. Pour cela, nous soutenons qu'une approche possible consiste à permettre la définition au sein du modèle de nouveaux squelettes algorithmiques et de nouveaux connecteurs et à permettre de ne choisir leurs mises en œuvres que lorsque les ressources d'exécution sont connues. Les contributions présentées dans ce manuscrit consistent en l'étude des modifications à apporter aux modèles de composants pour permettre cela, ainsi qu'en un prototype de mise en œuvre utilisé pour la validation du modèle résultant.

Support de squelettes à l'aide de la généricité et de la hiérarchie En combinaison avec la hiérarchie, le concept de généricité intégré aux modèles de composants permet de décrire des structures de composition dans lesquelles le traitement à appliquer ou le type des données traitées est un paramètres. Il s'agit en fait des concepts qui permettent la description de squelettes algorithmiques.

Connecteurs et hiérarchie Si le concept de connecteur permet de découpler la mise en œuvre du code fonctionnel des interactions entre ces codes, en présence de la hiérarchie l'approche habituellement utilisée pour l'intégrer aux modèles de composants pose problème. Elle présente une lacune qui peut empêcher de concilier performances optimales et interchangeabilité des mises en œuvres de composant. Nous avons donc proposé une nouvelle approche pour la description des interactions entre les composants basée sur le concept de *connexion ouverte* pour résoudre ce problème.

Plate-forme de mise en œuvre Afin d'utiliser le modèle ainsi défini, nous avons développé une plate-forme pour sa mise en œuvre nommée HLCM *implementation* ($HLCM_i$) qui est elle même développée à l'aide de composants. Elle offre une base pour la mise en œuvre des diverses spécialisations de HLCM. Pour cela, elle s'appuie sur une approche inspirée de l'ingénierie basée sur les modèles (*Model-Driven Engineering* – MDE). Nous avons utilisé cette plate-forme pour mettre en œuvre HLCM/CCM.

Évaluation Le modèle HLCM et plus particulièrement sa spécialisation HLCM/CCM ainsi que sa mise en œuvre basée sur $HLCM_i$ ont été évalués en les utilisant pour mettre en œuvre des applications synthétiques. Ces applications constituées de couplages de codes pouvant être parallèles à l'aide d'interactions par partage de donnée et par appel de méthodes parallèles ont été développée en s'appuyant sur HLCM/CCM et exécutées sur la plate-forme expérimentale *Grid'5000*. Le modèle HLCM et la plate-forme de mise en œuvre $HLCM_i$ ont aussi été utilisés pour permettre l'adaptation aux ressources d'exécution d'applications basées sur des opérations de communication collective hiérarchiques en collaboration avec Cristian Klein dans la lignée de ses travaux de master.

Perspectives

Ces travaux ouvrent plusieurs perspectives à plus ou moins long terme. À court terme, il s'agit avant tout de permettre une validation plus poussée du modèle et de sa mise en œuvre en les utilisant au sein d'applications de calcul scientifique existantes. À moyen terme il semble intéressant de proposer une étude des différentes approches qui peuvent être adoptées pour automatiser l'ensemble des choix effectués lorsque les ressources d'exécutions sont connues. À plus long terme, il serait intéressant de proposer des évolutions du modèle pour permettre d'y décrire un panel d'applications plus important en prenant en comptes des aspects qui ne sont pour l'instant pas gérées comme par exemple la généricité ou le partage d'éléments de mise en œuvre entre des éléments logiques distincts.

Utilisation au sein d'applications réelles La validation du modèle proposé dans ce manuscrit s'appuie principalement sur une utilisation pour la mise en œuvre d'exemples synthétiques d'applications qui posent particulièrement problème dans les modèles classiques. Une première utilisation dans le cadre d'une application réelle a été étudiée dans le cas de Dhico, mais elle s'appuie sur une spécialisation de HLCM dédiée à ce modèle. Une utilisation au sein de véritables applications permettrait de démontrer avec plus de force que le modèle proposé correspond effectivement aux besoins de celles-ci et qu'il ne possède pas de lacune rédhibitoires.

Des premiers contacts ont été avec pris avec l'équipe du professeur Kale de l'université d'Illinois à Urbana-Champaign (*University of Illinois at Urbana-Champaign* – UIUC) en vue de l'utilisation de HLCM avec Charm++ comme modèle sous-jacent au sein de l'application OpenAtom. Il s'agirait dans un premier temps de permettre l'utilisation de différentes mises en œuvres pour une opération de transposition et d'orthogonalisation de matrices en fonction des caractéristiques des ressources matérielles. Un autre aspect concerne le transfert de données avec redistribution entre deux parties de l'application dont une mise en œuvre efficace nécessiterait sans HLCM une modification profonde du code. La conception des bases pour l'utilisation de Charm++ comme modèle primitif a commencé et les besoins en termes de composants et de connecteurs ont été étudiés.

Amélioration des algorithmes de choix Le modèle que nous avons proposé permet de concevoir des applications décrites à un niveau d'abstraction qui leur permet de s'adapter aux ressources matérielles disponibles pour leur exécution. Cette adaptation repose sur un certain nombre de choix qui doivent être fait lorsque les ressources d'exécution sont

connues. Au cours de nos travaux, nous nous sommes contenté de choix manuels ou aléatoires qui ont permis d'évaluer le modèle en lui même. Afin de permettre une véritable adaptation automatique des applications aux ressources, il est toutefois nécessaire de concevoir des algorithmes permettant d'automatiser ces choix.

La difficulté pour la conception de ces algorithmes est double :

- le nombre d'opérateurs de composition envisageables et pour chaque opérateur, le nombre de mises en œuvres possibles sont illimités ; et
- ces opérateurs peuvent être utilisés conjointement au sein d'une même application en engendrant des interactions complexes.

L'approche classique consistant à s'appuyer sur un algorithme de planification possédant une connaissance profonde de l'ensemble des opérations de composition utilisées au sein des applications qui a été explorée dans le cadre de DHICO est donc difficile à mettre en œuvre dans un cadre plus général. Dans la mesure où il ne semble pas non plus envisageable de s'appuyer sur un modèle permettant de dériver le coût d'exécution d'une application en fonction de son déploiement, des solutions nouvelles doivent être trouvées.

Une évolution pourrait consister à attacher à chaque nouvel opérateur proposé (sous la forme d'un connecteur ou d'un composant générique) un algorithme permettant de faire le choix parmi ses mises en œuvres. Cette évolution se fait au prix d'une moindre possibilité de prise en compte des interaction entre les divers opérateurs utilisés au sein d'une même application et ne résout pas le problème des mises en œuvres supplémentaires pouvant être ajoutées après la conception de l'algorithme. Pour prendre en compte ce dernier point, une piste intéressante consisterait à associer aux mises en œuvres des informations sur leurs performances selon les métriques utilisées par l'algorithme de choix.

On pourrait éventuellement envisager une approche inspirée de l'auto-adaptation qui étudierait le comportement des applications au cours de leur exécution. Cette analyse en permettant de découvrir l'adéquation ou non des choix effectués initialement avec les objectifs permettrait de remettre en cause ceux qui ne semblent pas opportuns.

Finalement, un dernier point lié aux algorithmes de choix concerne leurs interactions avec les systèmes de gestions des ressources matérielles (*Resource Management Systems* – RMS). Ces systèmes sont en effet responsables d'effectuer des choix pour arbitrer l'attribution des ressources matérielles aux différentes applications s'exécutant sur une plate-forme partagée. Des travaux visant à étudier les solutions pouvant être proposées pour gérer les interactions entre les choix fait concernant l'application elle même et ceux concernant l'attribution des ressources matérielles ont lieu dans le cadre du projet ANR COOP.

Support de la dynamicité Le modèle que nous avons proposé propose de décrire un assemblage logique qui est transformé en fonction des ressources d'exécution disponibles pour donner l'assemblage qui est effectivement déployé. Ce modèle est donc restreint aux cas où l'assemblage n'évolue pas au cours du temps et une perspective serait de permettre de telles évolutions, c'est à dire d'introduire la dynamicité dans le modèle.

On retrouve par exemple ce besoins dans le cas des applications qui s'appuient sur un raffinement adaptatif de maillage (*Adaptative Mesh Refinement* – AMR). Ces applications, dont la mise en œuvre à l'aide de composants est étudiée dans le cadre des travaux de thèse CIFRE EDF de Vincent Pichon au sein de l'équipe projet INRIA GRAAL, s'appuient sur une évolution des algorithmes appliqués en fonction des données. En terme de composants, il s'agit de faire évoluer l'assemblage au cours de l'exécution de l'application en fonction des données qui y sont manipulées. Pour permettre la description et l'exécution de telles applications avec HLCM, il convient tout d'abord d'utiliser un modèle primitif permettant des évolutions de l'assemblage, mais ce n'est pas suffisant.

Il faut également étudier l'introduction d'une boucle de rétroaction pour permettre aux applications d'engendrer les évolutions de l'assemblage nécessaire en fonction des valeurs des données manipulées en leur sein. L'approche qui consisterait à laisser l'intégralité des choix être effectués par les applications en leur offrant simplement l'accès à un service de manipulation de l'assemblage est problématique dans le cadre de HLCM. En effet, l'assem-

blage de bas niveau qui est déployé ne correspond pas à l'assemblage de haut niveau qui était connu au moment de la description des applications. Pour résoudre ce problème, il est nécessaire que les différentes opérations de transformations puissent être ré-appliquées. Il s'agit en fait de faire évoluer les mises en œuvres de HLCM pour ne plus être de simples compilateurs mais de véritables exécutifs présents tout au long de la durée de vie des applications.

Deux approches au moins peuvent être envisagées pour la description des évolutions de l'assemblage qui ne sont pas incompatibles. Soit l'application garde le contrôle complet des évolutions de l'assemblage et effectue des requête de modifications de l'assemblage logique qui sont traduites par l'exécutif en terme d'assemblage de bas niveau. Soit le modèle est modifié pour intégrer des éléments de composition issus des modèles de flux de travail comme cela a été proposé dans le cadre des travaux sur le modèle de composition spatio-temporel STCM. La question de l'adéquation du concept de connecteur à ce nouveau type d'interactions doit être étudiée.

Partage d'éléments de mise en œuvre Le modèle HLCM propose la description d'assemblages logiques dont chaque élément (composant ou connexion) est mis en œuvre par un ensemble d'éléments offrant un niveau d'abstraction moins élevé jusqu'à arriver à des éléments primitifs directement mis en œuvre dans un modèle sous-jacent. C'est à dire que l'assemblage final peut être décrit sous la forme d'un arbre dont les nœuds internes sont les éléments abstraits et les feuilles les éléments concrets. Une conséquence de cette approche est qu'il n'est pas possible pour deux éléments abstraits distincts à un même niveau de la hiérarchie de partager un même élément concret de mise en œuvre.

Dans l'idée de permettre la réutilisation au sein d'une application d'une grande palette de code existant, il semble intéressant de permettre l'utilisation conjointe de plusieurs intergiciels de communications. Pour ne pas se limiter à un modèle sous-jacent qui intégrerait un nombre limité de tels intergiciels, une approche intéressante consiste à décrire les intergiciels eux même dans HLCM.

Au sein de ces intergiciels de communication, il faut toutefois noter qu'un ensemble de connexion logiques sont généralement gérées par un même exécutif. C'est même inévitable si on veut pouvoir multiplexer un ensemble de communications logiques au dessus d'une unique connexion de plus bas niveau. La forme hiérarchique des assemblages HLCM ne permet pas actuellement un tel partage. Il serait donc intéressant d'envisager l'introduction au sein du modèle de concepts permettant de partager certains éléments de mise en œuvre entre plusieurs éléments sans lien au niveau logique.

Les modèles de programmation classiques et notamment l'utilisation de bibliothèques partagées permettent un tel partage en identifiant les données constantes qui peuvent être partagées et les données qui peuvent être modifiées et sont donc spécifiques à une instance. Une autre piste intéressante est offerte par les assemblages du modèle de composants Fractal qui permettent le partage d'instances de composants entre de multiples composites et prennent donc la forme de graphes orientés acycliques (*Directed Acyclic Graph* – DAG) plutôt que d'arbres. Finalement, une dernière voie à explorer concerne les modèles de composition de services dans lesquels les compositions ne sont pas faites avec des instances bien identifiées mais avec l'instance qui offre le service le plus adapté à un besoins, instance pouvant donc être implicitement partagée.

Bibliographie

[1] Iso/isc dtr 19769 : Working draft, standard for programming language c++, August 2010.

[2] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming : Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[3] M. Aldinucci, H. Bouziane, M. Danelutto, and C. Pérez. Towards software component assembly language enhanced with workflows and skeletons. In *Joint Workshop on Component-Based High Performance Computing and Component-Based Software Engineering and Software Architecture (CBHPC/COMPARCH 2008)*, 14-17 October 2008.

[4] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, 20(2) :163–202, 2006.

[5] G. Antoniu, L. Bougé, and M. Jan. Juxmem : An adaptive supportive platform for data sharing on the grid. *Scalable Computing : Practice and Experience*, 6 :45–55, Nov. 2005.

[6] G. Antoniu, H. L. Bouziane, L. Breuil, M. Jan, and C. Pérez. Enabling transparent data sharing in component models. In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 430–433, Singapore, May 2006.

[7] G. Antoniu, H. L. Bouziane, M. Jan, C. Pérez, and T. Priol. Combining data sharing with the master–worker paradigm in the common component architecture. *Cluster Computing*, 10(3) :265 – 276, 2007.

[8] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' guide to netsolve v1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[9] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. P3l : A structured high level programming language and its structured support. *Concurrency Practice and Experience*, 7(3) :225–255, May 1995.

[10] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing : Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[11] D. Bálek and F. Plasil. Software connectors and their role in component deployment. In *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 69–84, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[12] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. Gcm : A grid extension to fractal for autonomous distributed components. *Special Issue of Annals of Telecommunications : Software Components – The Fractal Initiative*, 64(1) :5, 2009.

[13] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1) :39–59, 1984.

[14] E. Bohm, A. Bhatele, L. V. Kale, M. E. Tuckerman, S. Kumar, J. A. Gunnels, and G. J. Martyna. Fine grained parallelization of the car-parrinello ab initio md method on blue gene/l. *IBM Journal of Research and Development*, 52(1/2), 2007.

[15] H. L. Bouziane, C. Pérez, and T. Priol. A software component model with spatial and temporal compositions for grid infrastructures. In *Euro-Par '08 : Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 698–708, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] H. L. Bouziane, C. Pérez, and T. Priol. Extending software component models with the master-worker paradigm. *Parallel Computing*, 2010.

[17] G. Bracha. Generics in the Java Programming Language, Jul. 2004.

[18] E. Bruneton, T. Coupaye, and J-B. Stefani. *The Fractal Component Model, version 2.0.3 draft*. The ObjectWeb Consortium, Feb. 2004.

[19] E. Canot, C. de Dieuleveult, and J. Erhel. A parallel software for a saltwater intrusion problem. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. G. Plata, P. Tirado, and E. L. Zapata, editors, *Parallel Computing : Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *NIC*, pages 399–406, Department of Computer Architecture, University of Malaga, Spain, Sep. 2005. Central Institute for Applied Mathematics, Jülich, Germany.

[20] Eddy Caron and Frédéric Desprez. Diet : A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3) :335–352, 2006.

[21] Eddy Caron, Cristian Klein, and Christian Perez. Efficient grid resource selection for a cem application. In *RenPar'19. 19ème Rencontres Francophones du Parallélisme*, Toulouse, France, September 2009.

[22] B. L. Chamberlain, B. L. Chamberlain, D. Callahan, D. Callahan, H. P. Zima, and H. P. Zima. H.p. : Parallel programmability and the chapel language. *Int. J. high perf. computing*, 21 :231–312, 2007.

[23] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10 : an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05 : Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[24] M. Cole. *Algorithmic skeletons : structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[25] M. Cole. Bringing skeletons out of the closet : a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3) :389–406, 2004.

[26] Alexandre Denis. *Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, December 2003.

[27] Tarek El-Ghazawi and Lauren Smith. Upc : unified parallel c. In *SC '06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27, New York, NY, USA, 2006. ACM.

[28] David Chase Joe Hallett Victor Luchangco Jan-Willem Maessen Sukyoung Ryu Guy L. Steele Jr. Sam Tobin-Hochstadt Eric Allen. *The Fortress Language Specification Version 1.0*. Sun Microsystems, Inc., March 2008.

[29] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *OOPSLA*, pages 115–134, New York, NY, USA, 2003. ACM.

[30] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM : Parallel virtual machine : a users' guide and tutorial for networked parallel computing*. MIT Press, Cambridge, MA, USA, 1994.

[31] S. Gorlatch and J. Dünnweber. From Grid Middleware to Grid Applications : Bridging the Gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2005.

[32] Duncan Grisby, Apasphere Ltd., Sai-Lai Lo, David Riddoch, and AT&T Laboratories Cambridge. *The omniORB version 4.1 User's Guide*, July 2009.

[33] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *The MPI-2 Extensions*, volume 2 of *MPI : The Complete Reference*. The MIT Press, Cambridge, MA, USA, Sep. 1998.

[34] High Performance Fortran Forum. *High Performance Fortran Specification version 2.0*, janvier 1997.

[35] C. A. R. Hoare. Monitors : an operating system structuring concept. *Commun. ACM*, 17(10) :549–557, 1974.

[36] IEEE. *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1 : System Application : Program Interface (API) [C Language]*. 1996.

[37] L. V. Kale and S. Krishnan. Charm++ : Parallel programming with message-driven objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, pages 175–213. MIT Press, 1996.

[38] S. Lacour. *Contribution À L'automatisation Du Déploiement D'applications Sur Des Grilles de Calcul*. PhD thesis, Université de Rennes 1, IRISA, Rennes, France, Dec. 2005.

[39] Adrien Lèbre, Renaud Lottiaux, Erich Focht, and Christine Morin. Reducing kernel development complexity in distributed environments. In *EuroPar 2008*, Las Palma Spain, 2008.

[40] Sai-Lai Lo and Steve Pope. The implementation of a high performance orb over multiple network transports. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 157–172, London, UK, 1998. Springer-Verlag.

[41] Raúl López and Christian Pérez. Improving mpi support for applications on hierarchically distributed resources. In Franck Cappello, Thomas Hérault, and Jack Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 187–194. Springer, 2007.

[42] Elton Mathias, Françoise Baude, Vincent Cave, and Nicolas Maillard. A component-oriented support for hierarchical mpi programming on multi-cluster grid environments. *Computer Architecture and High Performance Computing, Symposium on*, 0 :135–142, 2007.

[43] S. Matougui and A. Beugnard. Two ways of implementing software connections among distributed components. In *OTM Conferences (2)*, pages 997–1014, 2005.

[44] D. McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 138–150, 1968.

[45] D. R. Musser and A. A. Stepanov. Generic programming. In *ISAAC '88 : Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation*, pages 13–25, London, UK, 1989. Springer-Verlag.

[46] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2) :1–31, 1998.

[47] Object Management Group. *MDA Guide Version 1.0.1*, Jun. 2003.

[48] Object Management group. *Common Object Request Broker Architecture Specification, Version 3.1*, Jan. 2008.

[49] Object Management group. *Common Object Request Broker Architecture Specification, Version 3.1, Part 3 : CORBA Component Model*, Jan. 2008.

[50] Open Service Oriented Architecture. *SCA Service Component Architecture : Assembly Model Specification Version 1.00*, Mar. 2007.

[51] OpenMP Architecture Review Board. *OpenMP Application Program Interface version 2.5*, mai 2005.

[52] C. Pérez, T. Priol, and A. Ribes. A parallel corba component model for numerical code coupling. In M. Parashar, editor, *GRID '02 : Proceedings of the Third International Workshop on Grid Computing*, volume 17 of *Lect. Notes in Comp. Science*, pages 88–99, Baltimore, Maryland, Nov. 2002. Springer-Verlag. Special issue Best Applications Papers from the 3rd Intl. Workshop on Grid Computing.

[53] Christian Pérez, Thierry Priol, and André Ribes. a parallel object model for high performance distributed systems. In *In Distributed Object and Component-based Software Systems, Hawaii Int. Conf. On System Sciences, IEEE*, 2004.

[54] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory : Concepts and systems. *IEEE Concurrency*, 4 :63–79, 1996.

[55] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf : A network based information library for global world-wide computing infrastructure. In *HPCN Europe*, pages 491–502, 1997.

[56] D. C. Schmidt. Guest editor's introduction : Model-driven engineering. *Computer*, 39(2) :25–31, 2006.

[57] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of gridrpc : A remote procedure call api for grid computing. In Manish Parashar, editor, *Grid Computing — GRID 2002*, volume 2536 of *Lecture Notes in Computer Science*, pages 274–278. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-36133-2_25.

[58] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *The MPI Core*, volume 1 of *MPI : The Complete Reference*. MIT Press, Cambridge, MA, USA, 2 edition, Sep. 1998.

[59] R. Srinivasan. Rpc : Remote procedure call protocol specification version 2, 1995.

[60] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[61] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.1*, novembre 2003.

[62] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2 edition, 2002.

[63] Gregory Zelesnik. *The UniCon Language Reference Manual*. School of Computer Science Carnegie Mellon University Pittsburgh, Pennsylvania 15213-3890, May 1996.

[64] K. Zhang, K. Damevski, V. Venkatachalapathy, and S. G. Parker. Scirun2 : A cca framework for high performance computing. *High-Level Programming Models and Supportive Environments, International Workshop on*, 0 :72–79, 2004.

Résumé

Les applications scientifiques posent un véritable défi de par leur complexité et la quantité de données qu'elles traitent. Leur exécution nécessite l'utilisation de ressources matérielles complexes et variées comme les super-calculateurs, les grappes et les grilles de calcul. Les modèles de composants logiciels et en particulier ceux qui proposent des schémas de composition adaptés au calcul à haute performance offrent une piste intéressante pour leur développement. Ces derniers permettent aux applications de s'abstraire des ressources d'exécution et de favoriser ainsi les performances à l'exécution sur une grande gamme d'architectures. Ces modèles restent toutefois limités à un ensemble restreint de schémas de composition. Il en résulte une multiplication des modèles dès que de nouvelles formes de composition apparaissent nécessaires, ce qui est dommageable à l'objectif de réutilisation. La complexité de modifier un modèle existant pour y intégrer de nouveaux opérateurs de composition défavorise aussi ces efforts. Cette thèse propose un modèle de composants logiciels appelé HLCM dans lequel de nouveaux opérateurs de composition peuvent être introduits sans modification du modèle. Ceci est rendu possible par l'introduction en son sein de quatre concepts : la hiérarchie, les connecteurs, la généricité et le choix de mise en œuvre. Afin de favoriser la réutilisation de l'existant, ce modèle est abstrait et il est rendu concret dans des spécialisations qui s'appuient sur les éléments primitifs de modèles existants. Au cours de ces travaux, nous avons étudié comment le concept de généricité pouvait s'appliquer aux modèles de composants et nous avons proposé une approche pour l'y intégrer. Nous avons étudié les interactions problématiques entre les concepts de connecteur et de hiérarchie et avons offert une solution s'appuyant sur une nouvelle approche pour la description des connexions entre composants. Finalement, nous avons proposé une approche pour la mise en œuvre de HLCM qui s'appuie sur des principes issus de l'ingénierie basée sur les modèles. Ces travaux ont été validés en développant un prototype de mise en œuvre de HLCM/CCM, une spécialisation de HLCM qui s'appuie sur le modèle de composant de Corba (CCM). Il a été utilisé pour décrire des interactions complexes entre composants (mémoire partagée et appels de méthode parallèles) et assurer une exécution efficace sur des ressources matérielles variées, notamment sur la plate-forme expérimentale Grid'5000.

Abstract

Scientific applications are a real challenge for programmers due to their complexity and the amount of data they manipulate. Their execution requires the use of complex and varied hardware resources such as supercomputers, computing clusters and grids. Software component models and specifically those that offer composition schemas dedicated to high performance computing offer an interesting approach for their development. They make it possible for applications to abstract themselves from hardware resources and to obtain high performances on a vast range of architectures. These models do however remain restricted to a limited set of composition schemas. This leads to a multiplication of models as soon as new composition forms are required. This is a problem for code reuse that is stated as one of the motivations behind software component models. The complexity of modifying existing software component models is also limiting these efforts. This thesis defines a software component model called HLCM in which new composition operators can be introduced without modifying the model. This is possible thanks to the introduction in HLCM of four concepts: hierarchy, connectors, genericity and implementation choice. In order to increase code reuse, this component model is abstract and it is made concrete in specialisations that are based on primitive elements of existing models. During this thesis, we have studied how the concept of genericity can be applied to software component models and we have proposed an approach to implement it. We have studied the problem that arises when combining the concepts of connectors and hierarchy and we have proposed a solution based on the novel approach for the description of interactions amongst components. We have also described an approach for the implementation of HLCM based on concepts originating in the field of model-based software engineering. These propositions have been validated by the development of a prototype implementation of HLCM/CCM, a specialisation of HLCM based of the CORBA Component Model (CCM). This implementation has been used to describe complex interactions between components (shared memory and parallel method calls) and to ensure an efficient execution on various hardware resources such as the Grid'5000 experimental platform.

N° d'ordre : D 10-31

Institut National des Sciences Appliquées de Rennes
20, Avenue des Buttes de Coëmes • CS 70839 • F-35708 Rennes Cedex 7
Tel : 02 23 23 82 00 - Fax : 02 23 23 83 96